Aleksi Luomala

MQTT’s Capabilities in an Unstable Network

Tieto- ja viestintätekniikka

Insinööri

Syksy 2018

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Tiivistelmä

Tekijä(t): Aleksi Luomala

Työn nimi: MQTT's Capabilities in an Unstable Network

Tutkintonimike: Insinööri (AMK)

Asiasanat: MQTT, Mosquitto, TCP/IP, Proxy, QoS, Quality of Service

Kyseisen insinöörityön toimeksiantaja oli Kalmar Cargotec. Yritys tarjoaa rahtisatamavälineitä, -toteutuksia sekä automatisoituja rahtisataman automatisointiratkaisuja. Työn tehtävänä oli selvittää MQTT-välittäjän soveltuvuutta yrityksen edeltävän viestintälaitteiston korvaajaksi. Tämä tuli selvittää testaamalla MQTT:n toimintaa simuloidussa, epäluotettavassa verkossa. Työssä tuli selvittää, onko MQTT:n oletuspersistointi riittävä takaamaan yhteyden palautuksen ja luotettavan viestien välittämisen tilanteissa, joissa yhteys voi katketa. Oli myös selvittävä, onko oletuspersistoinnin suorituskyky riittävä. Lisäksi haluttiin testattavan eri asiakasohjelman ja MQTT-välittäjän konfiguraatioiden vaikutusta suorituskykyyn. Suorituskyky ja persistoinnin toiminta eri QoS-luokilla tuli testata. QoS 0-luokka jätettiin pois testauksesta, sillä sen tärkeys yrityksen tarkoitusperiin koettiin olevan marginaalinen.

Työtä varten tuli asentaa ja konfiguroida MQTT-välittäjä, koodata asiakasohjelma, jonka avulla viestejä voitaisiin lähettää ja vastaanottaa MQTT-välittäjältä. Asiakasohjelman ja MQTT-välittäjän väliin tuli tehdä välityspalvelin, jonka avulla epäluotettavan verkon simulointi toteutettiin. Sen avulla viestien läpivienti pystyttiin ajastetusti estämään ja viestien toimittamista pystyttiin tutkimaan. Työn kannalta oli tärkeä tutkia tilannetta, jossa katkos ja viestien katoaminen tapahtui niin lyhyellä aikavälillä, ettei MQTT-välittäjä tai asiakasohjelma aikakatkaissut yhteyttä. Lisäksi tuli tutkia toimintaa tilanteessa, jossa yhteyden katkaisu tapahtui niin pitkällä aikavälillä, että MQTT-välittäjä sekä asiakasohjelma pystyivät itse toteamaan yhteyden olevan poikki.

Testit osoittivat, ettei asetusten muuttaminen merkittävästi vaikuttanut MQTT:n kykyyn palautua odottamattomasta yhteyden katkeamisesta. Viestien toimituksen nopeuden testaamisessa merkittävimmiksi asetuksiksi nousivat QoS-asetuksen sekä pysyvyysasetuksen muuttaminen. QoS-luokan muutoksella 5000 viestin toimitusnopeus saatiin 5,5 sekunnista 3,0 sekuntiin. Pysyvyyden aktivoiminen aiheutti toimitusajan nousemisen 27 sekuntiin. Vaikutukset olivat erittäin merkittäviä. Automaattinen tallennus ja pysyvyys yhdessä aiheuttivat testiajojen epäonnistumisen useita kertoja.

Yleisesti ottaen MQTT vaikuttaa sopivalta sen tarkoitettuun rooliin, vaikkakin pysyvyys ei tarjonnut odotettua suorituskykyä. On todennäköistä, että otettaessa MQTT käyttöön joudutaan pysyvyys tuottamaan itse, paremman suorituskyvyn takaamiseksi. Pysyvyyden ja automaattisen tallennuksen välinen toimimattomuus tuskin nousisi esteeksi.

Työn kaikki koodi tehtiin C:llä tai C++:lla. Kaikki koodi, joka ei ollut MQTT Paho C-asiakasohjelmaan suoraan yhteydessä, pyrittiin toteuttamaan C++:lla. Koodausympäristönä toimi Microsoft Visual Studio 2017. MQTT-välittäjänä työssä toimi Mosquitton toteutus. Työ toteutettiin Microsoft-käyttöjärjestelmälle.

Abstract

Author(s): Luomala Aleksi Henrikki

Title of the Publication: MQTT's capabilities in an unstable network

Degree Title: Bachelor of Engineering

Keywords: MQTT, Mosquitto, TCP/IP, Proxy, QoS, Quality of Service

This thesis was created for Kalmar Cargotec. The company offers equipment for cargo ports, port implementations and port automation solutions. The aim of the thesis was to examine the suitability of a MQTT broker as a replacement for the current messaging solution. This was to be determined by testing MQTT's behavior in a simulated, unstable network environment. It was important to examine how default persistence in MQTT would handle disconnections. The performance of the default persistence was also to be examined. Another area of interest was the impact to performance of different settings in client program and MQTT broker. Performance of MQTT and persistence setting's functionality needed to be tested on different QoS classes. QoS 0 testing was dropped due to company having a marginal need for it.

For testing a MQTT broker need to be installed and configured. A client program needed to be programmed, which was then used to send and receive messages through the MQTT broker. A TCP proxy was also needed in between a client and the MQTT broker, in order to simulate unstable network. By using a proxy the disconnection could be timed and controlled. For the thesis it was important to examine situation where the MQTT couldn't recognize the disconnection, as well as having a test case where the disconnection time was long enough for MQTT to realize there was a disconnection.

Tests showed that configurations had a small impact on how the MQTT broker was able to recover from a disconnection. QoS setting had a noticeable impact but persistence had by far the largest impact on the results.

Overall MQTT seems suitable for its intended role, even though persistence option didn't offer the performance that was expected. It's likely that if the MQTT was to be implemented, that they'd need their own solution for the persistence, to offer a more suitable performance for their needs.

All coding was done in C and C++. Everything that wasn't connected to MQTT Paho C-client was done in C++. Microsft Visual Studio 2017 was used as an integrated development environment. Mosquitto's solution of an MQTT broker was used in all tests and all tests were ran on Microsoft Windows 10 operating system.

Content

Abbreviations

QoS	Quality of Service
MQTT	MQ Telemetry Transport
OASIS	Organization for the Advancement of Structured Information Standards
ISO	International Organization for Standardization
TCP	Transmission Control Protocol
IP	Internet Protocol
SSD	Solid-State Drive

1 Introduction

The aim for this thesis was to examine if the MQTT standard provides sufficient tools and performance for Kalmar to integrate it as part of their software solution. Kalmar Cargotec sells terminal solutions and terminal equipment to ports around the world. In recent years there has been a push towards automation in this industry. The idea for this thesis came from within Kalmar as they had been looking to update one of their communication system in use. MQTT was considered as an alternative. I was tasked to test out MQTT's capabilities in an unstable network. People at Kalmar were mostly interested in what the default configuration had to offer in terms of stability and speed. It was essential for Kalmar that the equipment replacing their old system would be capable of delivering messages in a network where peers might occasionally lose connection. In a terminal environment it's possible that equipment is dropped out of the wireless network every now and then due to obstruction between it and the connection node. It's usually caused by stacks of cargo containers blocking signal. MQTT offers persistency by default which was of interest to people at Kalmar. They wanted to know how much deviation from the default configuration was required to suit their needs. If MQTT could offer a suitable solution that could be used out of the box, it would save development time as well as lessen the burden of maintenance.

To test these things a setup with two clients, publisher and a subscriber, a client and a TCP proxy was required. Publisher connected to a broker whereas the subscribing client connected to the TCP proxy. Which then connected to the MQTT broker. Publisher started publishing messages and after a couple of hundred messages TCP proxy was turned to block all incoming and outgoing messages. Proxy also discarded all messages it had received but hadn't delivered. After a set amount of time TCP proxy reconnected to MQTT broker if necessary and it was left to the client and the broker to sort out the situation with missing messages and broken connection. TCP proxy disconnection and reconnection times were documented along with every message's sent time and receive time. From those flight times and average flight times were calculated. Also a timeline was created which showed how many messages had been sent, how many received on a 0.5 second interval. It showed the amount of messages sent and received in total as well the amount of messages sent during the 0.5 second period. By comparing that data between runs with same and different settings, it was possible to determine which settings had an impact to performance and which didn't.

2 Theoretical background

As TCP and MQTT's functionality were both important for the subject of this thesis, the theory section covered what TCP is and how it functions. It also covers the most central parts of MQTT standard regarding the tests performed in the thesis.

2.1 TCP

Transmission Control Protocol is widely used protocol for data transmission. It provides error free, reliable and ordered data delivery. Before any data can be delivered, connection needs to be established. TCP connection uses ports to determine the end-points on a host system. Establishing connection between a client and a server uses three stage handshake. Client starts the handshake by sending its sequence number. Server responds by sending an acknowledgement as well as its own sequence number. Once the client receives those it answers with an acknowledgement message. Connection has now been established. Once the connection is up and running messages can be sent and received. TCP connection is capable of sending and receiving data through the same connection. Each TCP message has a header with a size between 20 and 60 bytes. This means that sending a byte you'd need to transfer at least 21 bytes of data [1]. Figure 1 below provides a depiction what the header of a TCP message looks like.

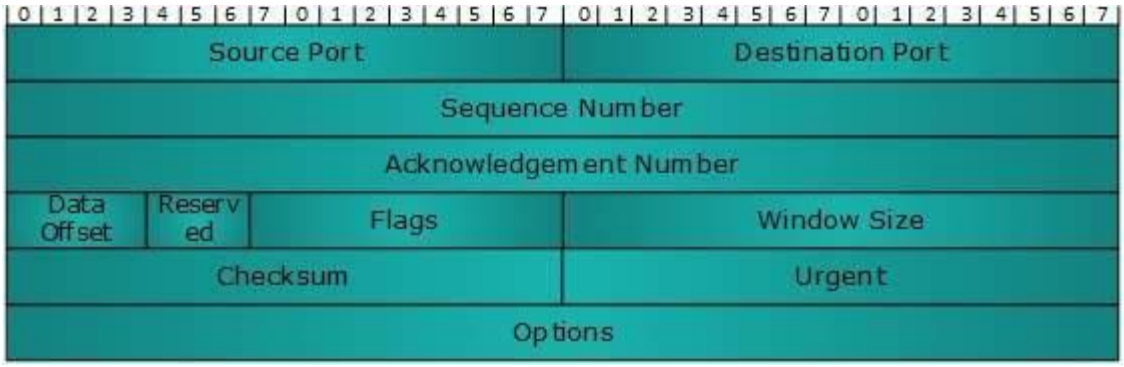


Figure 1. Representation of what a TCP header looks like [1].

When a client and a server start sending data over TCP, the connection starts off slow. This is because of the window size concept used in TCP connections. It means that a TCP connection manages bandwidth by limiting how many bytes can be sent at the time. The bandwidth can anything between 2 and 65 535 bytes. It can be increased or decreased during run time to control data flow. This prevents slower connections to crash

due to overload. TCP protocol is commonly used for websites, email or remote administration. Things that don't require reliable packet delivery but rely on fast, real-time interaction like multiplayer games over the internet, video streaming or Voice Over IP usually use User Datagram Protocol instead. It provides less latency as it doesn't need to wait for the packets to be organized into correct order [2].

2.2 MQTT

MQTT is a lightweight messaging protocol invented by IBM. It uses publish/subscribe structure and was developed for devices with constrains as well as for unreliable networks. It was also designed to cover use cases of high- and low-bandwidth. It has become a widely used protocol as it is well suited for "Internet of Things" development. Where connected devices are less powerful and connection bandwidth is often limited. On top of that it's also suitable for Machine to Machine (M2M) context. [3] MQTT protocol runs over TCP/IP network protocol [3, 4].

It was designed by Dr. Andy Stanford-Clark and Arlen Nipper in 1999. Dr. Andy Stanford-Clark was working for the IBM and Arlen Nipper was working for Arcom at the time [5]. In 2011 IBM and Eurotech announced their join participation in the Eclipse M2M Industry Working Group and donation of MQTT code to the proposed Eclipse Paho project [3]. During March 2013, MQTT was under the process of standardization. Finally during 2014, MQTT became part of the OASIS standard and later in 2016 it was accepted as part of the ISO tandard [6, 7]. Nowadays MQTT is now widely used [8, 9]. There are variety of different implementations that have been built on top of Stanford-Clark's and Nipper's creation [10].

Operating MQTT is simple. First MQTT broker must be running, after which client applications can connect to the MQTT broker. When connection is established, client's last will is given to the MQTT broker. Last will contains information of what the MQTT broker should do if the client disconnects ungracefully. The design of the MQTT means that separate clients don't have knowledge about each other. They don't know how many clients will be receiving their published messages. This allows good scaling for MQTT. Once MQTT broker and client have established a connection, client can then subscribe and publish to different topics. Client can receive and send data to topics simultaneously. Clients are also able to subscribe and publish to every topic by default. This behavior can be controlled in the MQTT broker's configuration file. If a client disconnects from the MQTT

broker gracefully, the MQTT broker will discard that client's last will and testament. If the disconnection was ungraceful, for example due sudden loss of connection, the MQTT broker will act according to the client's last will and testament. Last will might contain a message about the client's disconnection. The message is published within the topics the disconnected client was connected to [11].

2.2.1 Packets

Connect

Once the connection between an MQTT broker and a client is established, client sends its first packet, a CONNECT packet. Only one of these can be sent by a client. If anymore were to be sent by the client, the MQTT broker would need to consider these as protocol violations and disconnect the client [12, p. 23–30].

Connack

After client has sent the CONNECT packet and the broker has received it, the broker will send a CONNACK packet. If the first message isn't CONNACK or if client doesn't receive the message in a reasonable time, client should close the connection [12, p. 30–31].

Subscribe

SUBSCRIBE packet is sent to a broker whenever a client subscribes to one or more topics. The broker will send PUBLISH packets to the client regarding topics the client has a subscription to. SUBSCRIBE packets also carry the information about the maximum QoS for each subscription. The broker cannot send any messages from specific topic with a higher QoS, than what was specified for that topic in SUBSCRIBE message [12, p. 40–42].

Publish

A PUBLISH packet is sent by clients to a broker, which then delivers those packets to clients which are subscribed to the topics where the packet was published. PUBLISH message header contains info about packet type, if the packet is a possible re-delivery of an earlier message, QoS level and whether the message should be retained or not [12, p. 33–36].

Puback

When a sender publishes a message on QoS 1 setting, receiver will respond to the PUBLISH packet by sending a PUBACK. This informs the sender the message was delivered at least once [16, p. 32].

Pubrec

When a sender sents a PUBLISH packet at QoS 2, a PUBREC packet is sent by receiver as a response to inform the sender a PUBLISH packet was received [12, p. 37–38, 18].

Pubrel

When a sender receives a PUBREC packet, it informs receiver by sending a PUBREL packet. This informs receiver that the packet was delivered [12, p. 38–39, 13,].

Pubcomp

PUBCOMP is sent by the receiver to the sender when a PUBREL packet is received. This informs sender that receiver knows the message was received. Now both the receiver and the sender know the message was delivered successfully and only once [12 p. 39–40, 13].

Disconnect

When a client disconnects gracefully, it sends a DISCONNECT packet to a broker. It's the last message the client sends, and it must close its network connection afterwards. The broker will discard last will from the client which sent the DISCONNECT packet [12, p. 49].

2.2.2 Quality of Service

Quality of Service (QoS) has 3 different levels. The higher the level of QoS, the more overhead messages have. Whenever a client subscribes to a topic, it informs broker about the QoS it wants to receive messages with. Client A might send a message with QoS 2, but since client B subscribed with QoS 1, the broker will send the message it received from client A to client B with QoS 1. Figure 2 below presents how QoS levels differ from each other in terms of messages sent to deliver a single PUBLISH message.

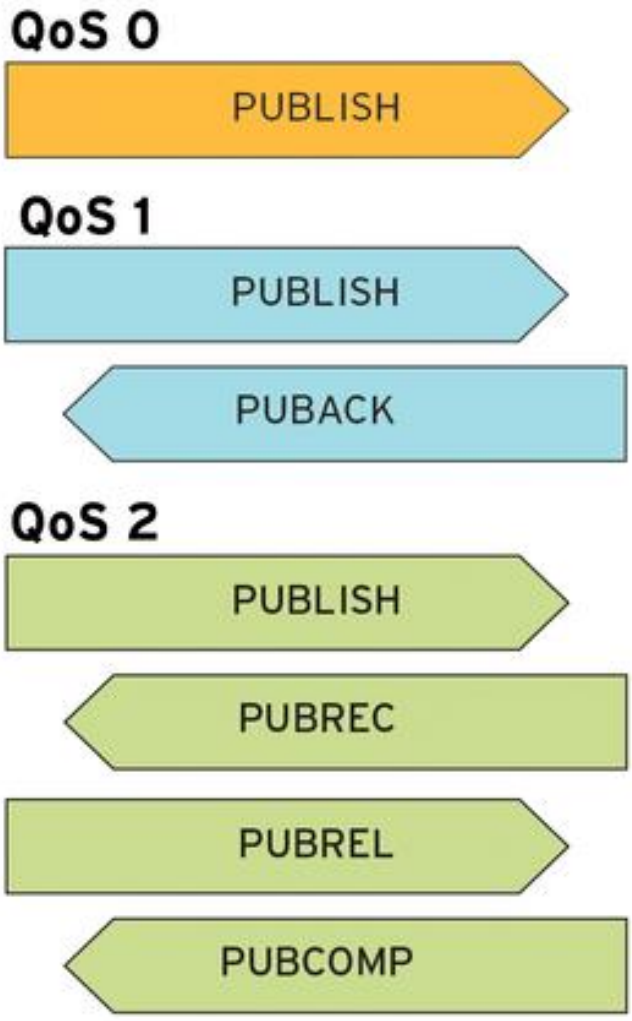


Figure 2. Communication between a client and a server presented on different QoS levels when one message is published [14].

In QoS level 0, a message will be delivered at most once. The message is sent alone without any responses. It might not arrive at all and clients won't be informed about successful arrival of any message.

In QoS level 1, a message will arrive at least once. Before sending a message, a sender will store it temporarily. Once the message is sent, sender waits for PUBACK packet to arrive. If the sender doesn't receive PUBACK packet quickly enough from the receiver, it will send the message again. Because of this design it's possible for the receiver to receive the same message multiple times.

QoS level 2 ensures a message will arrive only once. At first a sender stores a PUBLISH message it's about to send. Once that's done, the message is sent. When a receiver receives the message, a reference to the packet identifier is stored. The receiver then

responds by sending a PUBREC message. Once sender receives the PUBREC message, it can discard the PUBLISH message as it has been delivered successfully. PUBREC is then saved by the sender, after which a PUBREL message is sent to the receiver. Receiver can safely discard all previous messages from this exchange as it receives the PUBREL. Receiver then sends the last message called PUBCOMP. At this stage the packet identifier from the PUBLISH message can be discarded. Both now know the message was sent successfully and only delivered once [15].

2.2.3 Retain

For each message in MQTT there's an option to retain it. If the retain flag is set to 1 when the message is sent, it tells the broker the message must be saved. Broker will then save message's content and the QoS of the message. Retained messages could then be delivered to new subscribers subscribing to the topic where the retained message was published in. A message with QoS 0 and retain flag set to 1 indicates to a broker that it needs to discard any previously retained messages. It should store the newly arrived message, but may choose to discard it at any time.

When a broker is sending a PUBLISH message to a client, it's required for the broker to set retain flag as 1 when it's send because of a new subscription. Otherwise it must be set to 0 by the broker.

Whenever a broker receives a PUBLISH message with retain flag set to 1, but payload containing zero bytes, it will be sent to the subscribers in the same topic where the message was published. It will also cause broker to discard any retained messages and prevent any future subscribers for receiving retained messages from the topic where the message was published. Furthermore a message with a payload of zero bytes will not be stored by the broker as a retained message. Messages without the retain flag are not stored by the broker nor do they replace any stored retained messages [12, p. 34–35].

3 Setting up test environment

To test how MQTT functions in an unreliable network, three parts were required. Clients to both subscribe and publish, a TCP proxy to simulate the disconnections and data logs to analyze test runs. Following chapters go through the structure and functionality of each of the components.

3.1 TCP proxy

TCP proxy that was used for this thesis was designed to simulate an unreliable connection. By simulating bad connection, instead of using one, it ensured the results received from testing were consistent. A simple application was created which worked as a bridge between a client from client application and MQTT broker. Bad connection was simulated by providing disconnection duration for TCP proxy. Information about disconnection and reconnection was delivered via another connection between the client and the proxy. Two different disconnection cases were examined.

In the first case connection went offline for a short duration. The time TCP proxy was disconnected was less than the time it takes a broker or a MQTT client to notice either one had disconnected. Ideally MQTT system should have been able to recover from it automatically and without any extra configuration. for this test case TCP proxy stopped delivering messages for five seconds.

In the second test case the connection went offline for long enough that the broker and a MQTT client were able to notice the disconnection. Here communication between the two should have recovered automatically as well. Disconnection of the client was done using Proxy with 30 second disconnection duration. All the data from each test run was saved into a log file which was then imported to Microsoft Excel where graphs were drawn based on the data that was gathered.

3.1.1 Structure

Proxy applications consisted of three parts. Each part ran in its own thread. First one handled MQTT messages arriving from and going to the connected client, the second

handled MQTT messages arriving from and going to the MQTT broker and last one for the communication between the connected client and the proxy. When Proxy was started, it waited for a connection from the client. First the command connection was established. It was used for communication between proxy and the client. It was followed by the client connecting to the socket which it will use to receive and send data to the MQTT. Finally Proxy establishes connection to the broker.

The part handling messages arriving from and going to the client checks whether or not the connection is disabled. If it's disabled, it checks if the vector holding data coming from client is empty. If not, it's cleared. If the connection is enabled, socket for client is checked for messages. If there is messages coming from the client, the incoming data is saved to a vector which holds data coming from the client. Then it's checked if messages have been received from the broker. If so, the first one is sent to the client.

Messages going between proxy and broker are handled the same way. First connection's status is checked. If it's disabled, vector holding messages coming from the MQTT broker is cleared if it's not empty. Then the socket for broker is received for any incoming messages. Those messages are being held in a vector for messages coming from MQTT. Afterwards the first message arrived from client is passed to broker. During reconnection the socket would reconnect to the broker, in case it has terminated the connection.

Last part is responsible for communication between the proxy and a connected client. It checks if it needs to relay a disconnection. If so, it sends a message informing about the disconnection. The message also contains the time when the disconnection happened. It would then wait for an answer from the client. Only after the acknowledgement arrived it would actually "disconnect". Meaning it wouldn't pass messages between the broker and the client. If reconnection was required from the command handler, similar steps would be taken. Message would be sent with the time of reconnection and then it would wait for acknowledgement. Once acknowledgement arrived it would actually "reconnect" Meaning to pass messages between the broker and the client. The client would also actually re-establish the connection between the client and the broker. Figure 15 below presents how TCP proxy's threads are connected to other systems when a test was ran.

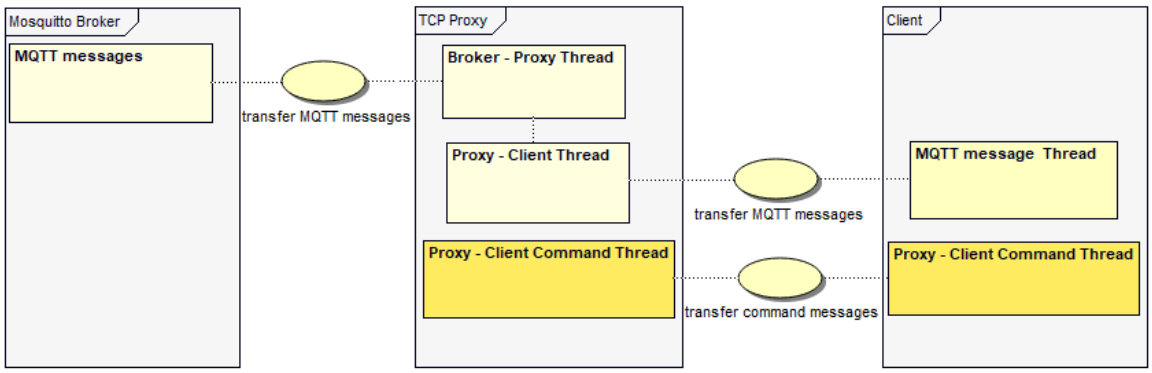


Figure 15. Representation of how TCP Proxy's threads were connected in the test setup.

3.2 Test client

Client was created as a separate project from the TCP proxy. Client side was built so only one client application was required to be launched at the time. Inside the client application two client objects, publisher and subscriber, were created with their settings listed before them. Client objects' settings were combined so the same parameters were applied to both of the client objects. This allowed changing settings easily and made sure they were applied to both client objects. Once both client objects were created their settings were initialized, two threads were started, one for each client object. The struct containing corresponding client's settings was passed to the thread with the client object.

Inside a thread a client object was first initialized. MQTTAsync was created for both, but since the subscriber needed to connect to proxy, its port was set to predefined port which the Proxy was listening. MQTTAsync callbacks were set for connection lost, message arrived and delivery succeeded. The client object was passed as context to the callbacks, so the object itself could be accessed. MQTTAsync disconnect options and connect options were set as well.

Once the initialization was ready, each object proceeded to connect to the broker. Publisher connected directly to the MQTT broker, whereas subscriber connects to TCP proxy's port. Once the subscriber connected and sent its first message, TCP proxy then connected to the broker, and forwarded the message it had received. Lastly a thread for commands was started. Since TCP proxy doesn't process MQTT messages it receives and thus doesn't know what kind of data and messages are being passed through the TCP proxy, second connection for communication between client object and TCP proxy was required. This also ensures the communication between the TCP proxy and client object didn't interfere with the communication between client object and the broker.

Once all the connections were established, the publisher proceeded to publish messages. For connection tests 100ms delay between messages was applied to simulate real world behavior of a device running as a MQTT client. Once the publisher sent all the messages, it disconnected and joined the client application's main thread. Since Paho's C client's asynchronous version uses callbacks when it receives messages, subscriber needed to do nothing but wait. A timeout timer was added to the subscriber's waiting loop to ensure it exits if messages were lost during the test. If one or more messages were lost, the thread waited until its timeout timer ran out and would then proceed to exit. If every message arrived on time, the subscriber exited right after the last message. On exit clients connected to TCP proxy would notify their command threads and wait for them to join before proceeding to disconnect from the TCP proxy.

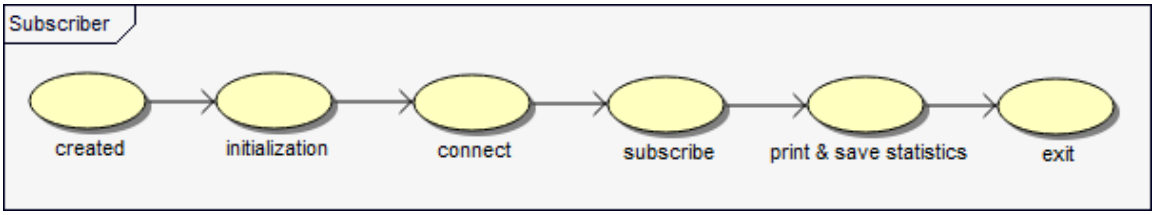


Figure 16. Subscriber client object's phases during a test run.

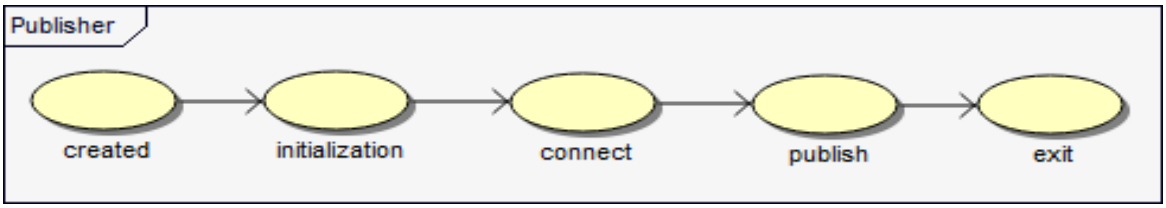


Figure 17. Publisher client object's phases during a test run.

3.3 Data logs and data representation

Data from each test run was saved to a log file. It was saved in a format that was easy to import to Microsoft Excel. Once test results were imported, charts were created. When that was done the data could be easily compared to other test results. Not every chart was created for every result. For disconnection tests following charts were created; connection status, messages sent per half a second, messages received per half a second, messages at broker, messages sent, messages received and average flight times. For throughput tests, following were created: messages sent, messages received, message flight time and messages received per half a second.

3.3.1 Logging data

Data logged during tests consists of four sections for each test run. The first section contains message's order number, first message having the number one, second having the number two and so on and so forth. Messages' sent and receive times were also contained there. These were saved with microsecond precision and system time was used to save the date time part of the time. Format for sent and receive times was hh.mm.ss:us, e.g. 15.38.48:945064. Time elapsed since the first message was sent was saved in date time format as well as in seconds. This meant the time that had passed since the first message was sent and the current message was received. Accuracy for recorded times was in microseconds. The same was then done to the first message received time. Then flight times were calculated from the time the message was sent and when it was received. This was stored with microsecond accuracy as well but represented in milliseconds, e.g. 12.594ms. Message order number, flight time and time elapsed since first message was sent ended up being the most vital pieces of information from this section. Figure 3 below shows what the data looked like before it was imported to excel.

```
;message_number; 0001 ;message_sent_time; 18.19.47:813623 ;message_arrival_time; 18.19.47:860750 ;time_el
;message_number; 0002 ;message_sent_time; 18.19.47:924859 ;message_arrival_time; 18.19.48:125112 ;time_el
;message_number; 0003 ;message_sent_time; 18.19.48:064554 ;message_arrival_time; 18.19.48:279114 ;time_el
;message_number; 0004 ;message_sent_time; 18.19.48:262359 ;message_arrival_time; 18.19.48:422088 ;time_el
;message_number; 0005 ;message_sent_time; 18.19.48:416968 ;message_arrival_time; 18.19.48:569199 ;time_el
;message_number; 0006 ;message_sent_time; 18.19.48:561064 ;message_arrival_time; 18.19.48:719725 ;time_el
;message_number; 0007 ;message_sent_time; 18.19.48:700807 ;message_arrival_time; 18.19.48:865150 ;time_el
;message_number; 0008 ;message_sent_time; 18.19.48:866230 ;message_arrival_time; 18.19.48:906862 ;time_el
;message_number; 0009 ;message_sent_time; 18.19.48:980695 ;message_arrival_time; 18.19.49:180906 ;time_el
;message_number; 0010 ;message_sent_time; 18.19.49:163956 ;message_arrival_time; 18.19.49:377354 ;time_el
;message_number; 0011 ;message_sent_time; 18.19.49:347572 ;message_arrival_time; 18.19.49:536185 ;time_el
;message_number; 0012 ;message_sent_time; 18.19.49:519569 ;message_arrival_time; 18.19.49:685972 ;time_el
;message_number; 0013 ;message_sent_time; 18.19.49:675148 ;message_arrival_time; 18.19.49:819552 ;time_el
```

Figure 3. Part of the message data which was stored into a log file after every test run.

Second section of test run data was interval based. Time step used for each test was 0.5 seconds. This section contained information about the total amount of messages sent and received, messages sent and received during the 0.5 second period and how many messages were at broker. All messages that had been sent but hadn't arrived were considered to be at the broker. This would introduce some form of inaccuracy since it didn't consider messages that had been sent but hadn't yet reached the broker. This inaccuracy was considered not to be an issue since the interest was of trends in behavior of multiple messages and average performance of multiple messages, rather than behavior of individual messages. Then the status of connection was also stored for each time step. At last the time step was saved as the last piece of information. Figure 4 shows what the data looked like once it was imported to excel. It is noteworthy that the first message was excluded from the 0.0 second mark so all the graphs start from 0. The first message was instead added to the 0.0-0.5 second interval result. This inaccuracy doesn't result in anything meaningful and was done for the sake of making cleaner graphs.

total_messages_received	0	messages_delivered	0	messages_sent	0	total_messages_sent	0	Messages_at_broker	0	connection_status	1	time_step	0
total_messages_received	5	messages_delivered	5	messages_sent	4	total_messages_sent	5	Messages_at_broker	0	connection_status	1	time_step	0.5
total_messages_received	9	messages_delivered	4	messages_sent	5	total_messages_sent	10	Messages_at_broker	1	connection_status	1	time_step	1
total_messages_received	14	messages_delivered	5	messages_sent	4	total_messages_sent	14	Messages_at_broker	0	connection_status	1	time_step	1.5
total_messages_received	18	messages_delivered	4	messages_sent	4	total_messages_sent	18	Messages_at_broker	0	connection_status	1	time_step	2
total_messages_received	22	messages_delivered	5	messages_sent	4	total_messages_sent	22	Messages_at_broker	0	connection_status	1	time_step	2.5
total_messages_received	27	messages_delivered	4	messages_sent	6	total_messages_sent	28	Messages_at_broker	1	connection_status	1	time_step	3
total_messages_received	32	messages_delivered	4	messages_sent	4	total_messages_sent	32	Messages_at_broker	0	connection_status	1	time_step	3.5
total_messages_received	36	messages_delivered	4	messages_sent	5	total_messages_sent	37	Messages_at_broker	1	connection_status	1	time_step	4
total_messages_received	41	messages_delivered	5	messages_sent	4	total_messages_sent	41	Messages_at_broker	0	connection_status	1	time_step	4.5
total_messages_received	45	messages_delivered	4	messages_sent	5	total_messages_sent	46	Messages_at_broker	1	connection_status	1	time_step	5

Figure 4. Test run statistics with a 0.5 second timestep.

Third section contained settings which were used for clients while testing. It also had a quick summary of the test results. In the figure 5 "total on wire" represents the amount of messages received. Total flight time was the amount of milliseconds spent in total by the messages to arrive. Time spent on average, with the total flight time were not all that useful information when data was logged for tests with disconnection, since the flight time contained the disconnection time on some messages. If a message took a 2.5ms in air on average and there was a 30 second disconnection time, the first message to arrive would have around 30002.5ms flight time. There was also information about when the first and the last message were received. Total time spent receiving messages was also logged. This could be used to determine that the run was successful, since the disconnection time was known, as well as the time delay between each message published. Proxy disconnect time and proxy reconnect times were logged in case it was required to do a sanity check on some message in the first section. This part served a purpose of being a sanity check for each test run. If messages were missing or it would have taken suspiciously long for a test to run it could be seen easily by checking this section.

Client settings were below the test run statistics. . QoS was either 1 or 2, depending which one was being tested. Clean session, retain and automatic reconnect used 1 or 0, where 1 means on and 0 means off. Keep Alive Interval was either 20 or 10 for the tests. Max retry interval was either 20 or 2 and min retry interval was either 10 or 1. These are supposed to represent seconds. For min retry interval value was either 10 or 1 and these were supposed to represent seconds as well. Persistence was either DEFAULT for default persistence or NONE when persistence was turned off. Reliable was either TRUE or FALSE depending whether it was on or off. These options are show in a result file once it was imported to excel down below.

total on wire	1000	
total_flight_time	825937	ms
time_spent_on_average_to_deliver_a_message	825.937	ms
first_message_received	15.37.42:671331	
last_message_received	15.39.24:018095	
total_time_spent_receiving_messages	00.01.41:346764	
proxy_disconnection_time	Wed May 16 15:38:10 2018	
proxy_reconnection_time	Wed May 16 15:38:15 2018	
QoS	1	
Clean_Session	0	
Keep_Alive_Internal	10	
Retain	1	
Automatic_Reconnect	1	
Max_Retry_Interval	20	
Min_Retry_Interval	10	
Retry_Interval	1	
Persistence	DEFAULT	
Reliable	TRUE	

Figure 5. Test run summary and client settings were saved with the rest of data gathered from a test run.

Fourth section was assembled from the flight times in the first section. When disconnections were being tested, the last 300 messages were used to calculate average message flight time and median flight time. Reason behind this was that the flight times during disconnection would render any results gathered meaningless. On top of that the start of the messaging did occasionally start off really slow. On some occasion the first 100 messages might have had flight times between 20ms and 100ms whereas for the rest of the run flight times would hover from 2.5ms to 10.0ms. This behavior was expected to be due to TCP connection being slow at the start. For 30 second disconnection, message delivery returned back to normal at around 600 messages. To eliminate effects from disconnection,

test was allowed to send 100 messages before taking results. It was kept the same for 5 second disconnections as its 300 messages were enough to indicate the performance. From the 300 messages, 3% of the highest flight times were then removed. This was done in order to eliminate random performance spikes which might or might not appear in the results. They might be caused by something non test related and were considered not to represent the average performance. When throughput was tested, all messages were included in the results. Absolute deviation from average and median flight times for each message was then calculated. From those results average deviation from average flight time and average deviation from median were calculated afterwards. These four values gave an idea about the expected performance. Figure 6 shows what the results of this might look like.

average flight time	median flight time		
2.327986	2.337		
Average of mean absolute deviation(average)			
0.068925			
Average of mean absolute deviation(median)			
0.068918			
97% of last 300 flight Times*	absolute distance from average(	absolute distance from median(ms)	
2.116	0.211986	0.221	
2.131	0.196986	0.206	
2.151	0.176986	0.186	
2.161	0.166986	0.176	
2.161	0.166986	0.176	
2.163	0.164986	0.174	
2.169	0.158986	0.168	
2.174	0.153986	0.163	
2.175	0.152986	0.162	

Figure 6. For tests which had disconnection, 97% of the last 300 messages were used to determine average, steady performance.

3.3.2 Graphs

Gathered data represented how a client with given settings would perform in the expected environment. Clear trends of slow or uneven delivery were both signs of poor performance. For test runs where disconnection was part of the test, following graphs were drawn; connection status, messages per half a second, messages received per second, messages at broker, messages sent, messages received message flight time before disconnection, message flight time after disconnection and average message flight time.

Those that had seconds in the bottom counted time from the moment the first message was sent. All graphs mentioned above had a 0.5 second time step.

Connection Status graph depicted connection status of the proxy. These graphs had connection status on the left and the time passed at the bottom. Whenever proxy was passing messages, it was shown as one, whereas when it didn't pass messages through its connection status was presented as zero, as it was considered to have disconnected. Figure 7 below presents a connection status graph from one of the runs as an example.

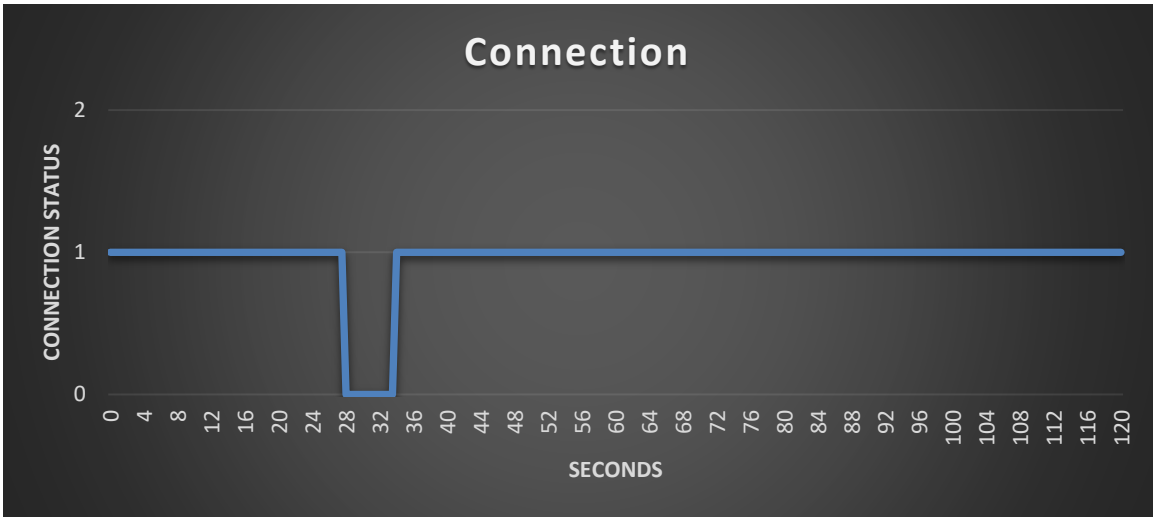


Figure 7. Test run graph to present how the graphs looked like.

In messages sent per half a second-graph had messages on the left and the time passed at the bottom. Line started at zero. This was because the first message sent was included into the 0.0-0.5 second time step, not 0.0 second timestep. This inaccuracy was introduced to make graph's line start from zero, which made it cleaner looking. Figure 8 below shows a typical graph that was received when test results were turned into a graph.

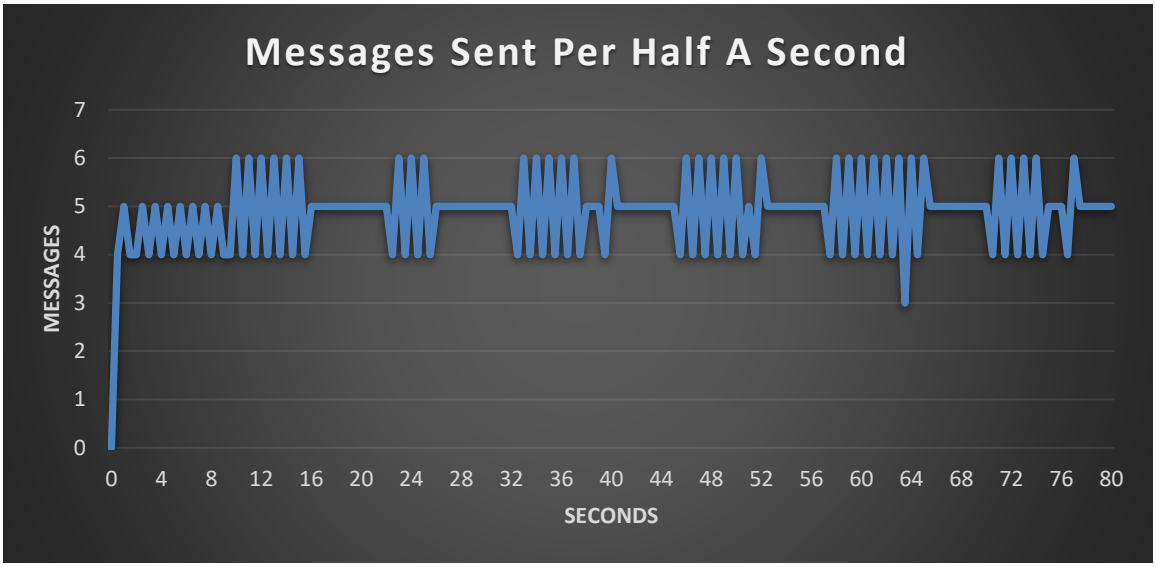


Figure 8. A shortened version of a messages per half a second graph.

Messages received per half a second-graph had the number of messages received on the left and the time passed in seconds at the bottom. These graphs were used to determine how quickly the broker was able to deliver messages once the connection was re-established. Figures 9 and 10 represent examples what the graphs looked like.

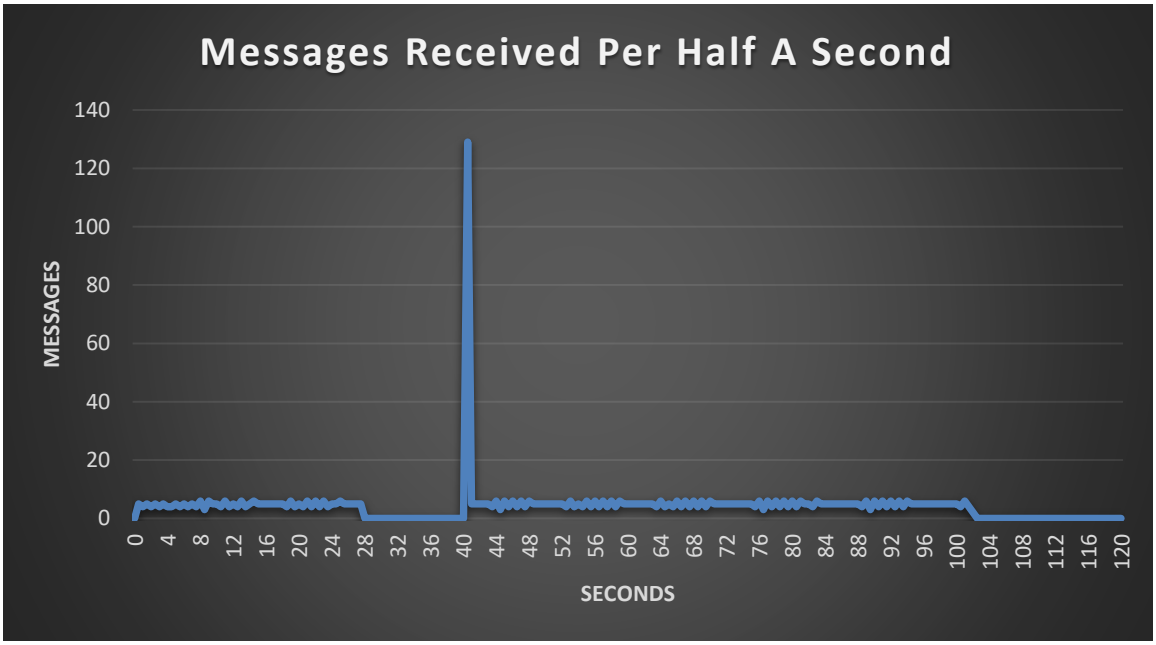


Figure 9. Results from a 10 second disconnection time.

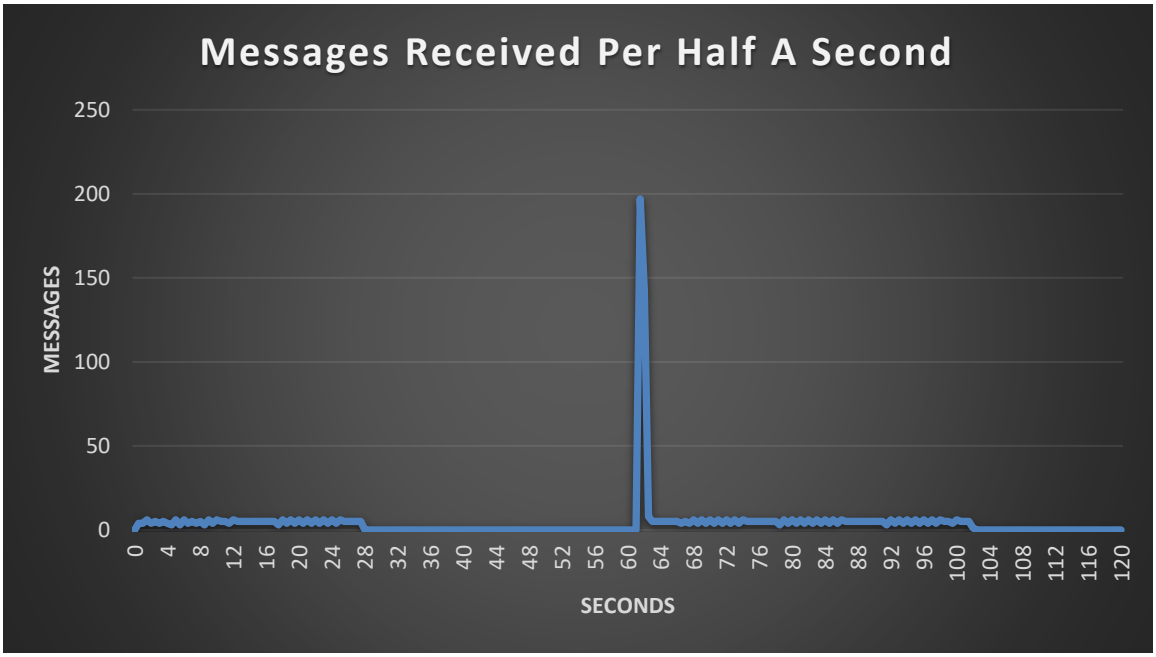


Figure 10. Results received from a 30 second disconnection time.

Messages at broker-graph depicts how messages pile up at the broker's side when the subscribing client is disconnected. This was used in combination with the previous graph when examining how quickly the broker was able to deliver messages when a connection was re-established. Figure 11 below shows an example of such graph.

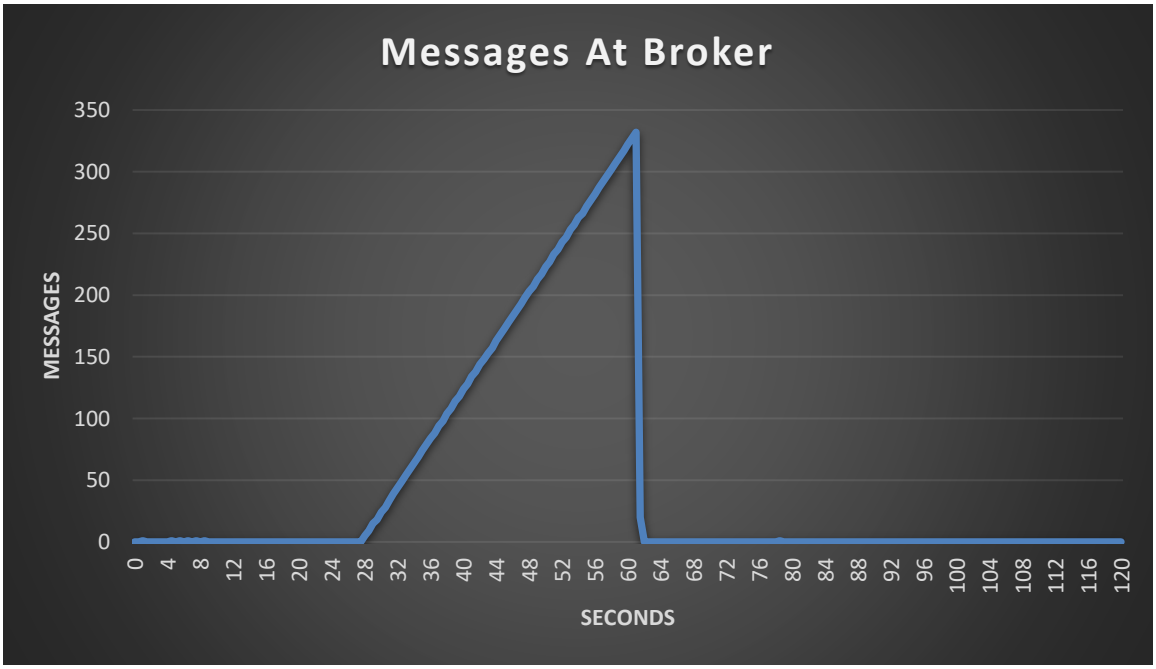


Figure 11. Graph represents how messages piled up at broker when subscribing client was disconnected.

Messages sent-graph had the total amount of messages sent on the left and time passed at the bottom of the graph. This was merely a sanity check when disconnection behavior was tested, and only became useful when throughput was being tested. Figure 12 below shows what messages sent-graph looked like when disconnection behavior was tested.

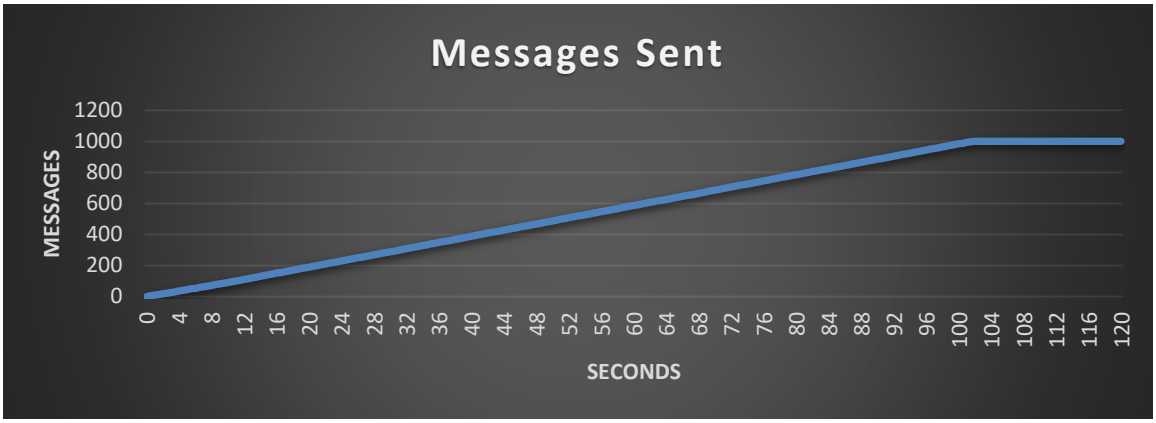


Figure 12. This graph was received when disconnection functionalities were being tested.

Messages received-graphs had number of messages received by the subscriber client on the left and the time passed at the bottom. These were mostly used when throughput was being tested but provided the similar data as messages at broker-graphs when disconnection time was being tested. Figure 13 is from a run with 30 second disconnection.

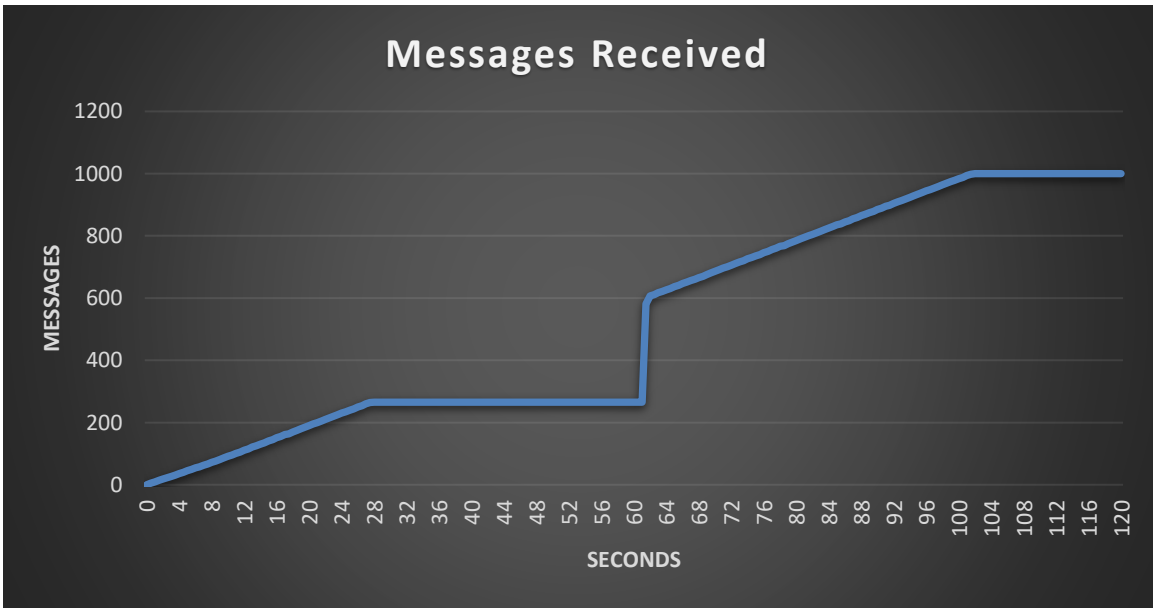


Figure 13. Messages received -graph from one of the runs with a disconnection.

Flight time averages graphs were used to compare performance between runs. They had milliseconds at the right and four pillars to represent 2 ways of measuring the average

flight time, and two ways of representing the deviation from the average value. First one was averaging all flight times. The second pillar represents the median flight time. Third one represented how much flight times deviated from the average flight time. The fourth pillar showed the average deviation from the median flight time. Figure 14 below shows an example result received during one of the runs.

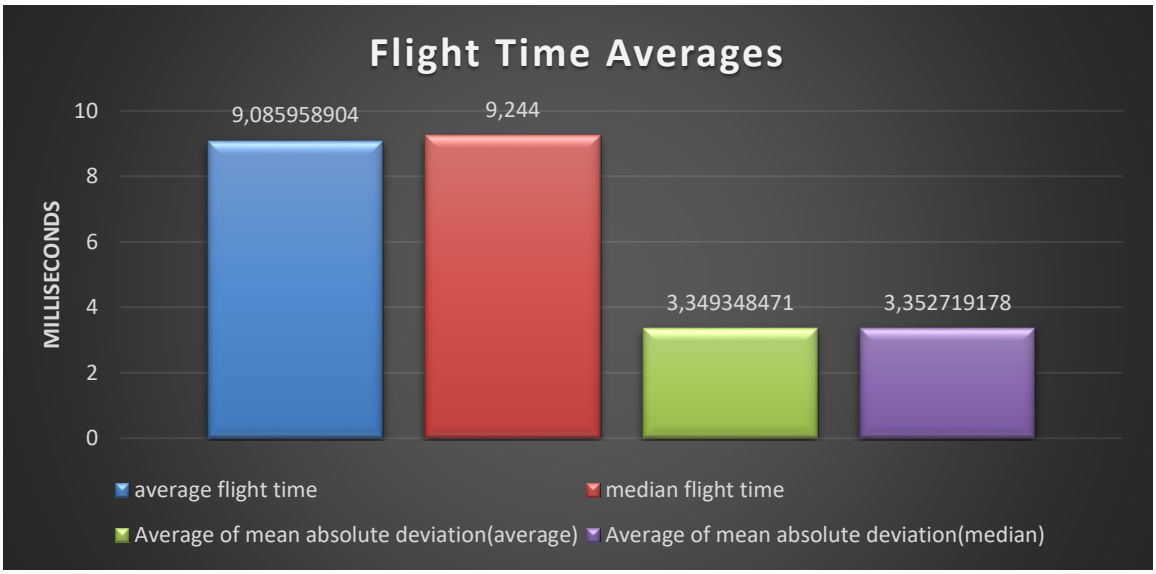


Figure 14. Deviation in flight times was calculated from both, median and average flight times. This gave more precise results compared to only using one or the other.

4 Testing

Testing setup for recovering from disconnection consisted of three main components; MQTT Client, TCP Proxy and MQTT broker. There were two MQTT client objects running inside the MQTT Client, publisher and subscriber. Publisher sent messages to the broker whereas the subscriber was connected to the TCP Proxy, which in turn was connected to the broker. Subscriber was subscribed to the topic where publisher client object published messages. After publisher had published 250 – 400 messages, TCP Proxy disconnected. Disconnection time was either 5 or 30 seconds. During the disconnection publisher kept publishing messages. After the disconnection time was over, TCP Proxy reconnected to the broker. Afterwards it was left to the broker and the subscriber client object to restore their connection. Once they had restored the connection, the remaining messages were delivered to the subscriber. This meant the messages stored to the broker during the disconnection plus the remaining messages. Total count for published messages was set to 1000. This ensured stable communication before disconnection and after disconnection, leaving enough space for the communication to return to normal after reconnection.

First the MQTT broker was started. Next TCP Proxy was started. Client application was the last application which was started. Inside the Client application Subscriber and Publisher client objects were launched and each ran inside their own thread with provided settings. Settings were the same for both client objects. Subscriber client then connected to the TCP proxy. At the same time Publisher client established connection with the MQTT broker. Publisher waited for 7.5 seconds, letting Subscriber client to establish connection with the TCP Proxy as well as letting the TCP Proxy to connect to the broker. Then Publisher client object started publishing messages. TCP Proxy was set so it had 35 seconds after creating all worker threads before it “disconnected” from the broker. This allowed communication to start and run stabile for a little while. Fluctuation in the number of messages published during this period didn’t matter, since the important part was when the TCP Proxy reconnected to the broker. After “disconnecting” from the broker, TCP Proxy then waited either 30 seconds or 5 seconds, depending which case was being tested. After the time was up, the TCP Proxy reconnected to the broker. When connection was re-established, the TCP Proxy sent a command packet to the Subscriber client object, containing a timestamp of the reconnection time. Once publisher was done publishing, it disconnected. Subscriber waited until it had received all the messages, or until 220 second had passed. When subscriber received all messages it then printed them onto a con-

sole window and also wrote them to a log file. Once that was done subscriber disconnected from the TCP Proxy. The TCP proxy ran until its 240 second timer ran out. Then it disconnected from MQTT broker and shut down.

Client settings:	
Quality of Service	1 and 2
Clean Session	false
Keep Alive Interval	20 and 10
Retain	1
Automatic Reconnect	1
Max Retry Interval	2, 20 and 40
Min Retry Interval	1, 10 and 20
Retry Interval	1
Persistence	default
Reliable	true and false
Broker settings:	
Max Inflight Messages	1 and 100
Persistence	true and false
Persistence: Autosave Interval	10
Max Queued Messages	1000
Proxy settings:	
Disconnection time	5 and 30 seconds

Table 1. Settings tested for both test case 1 and test case 2.

4.1 Test case 1: not realizing connection was interrupted

When testing short disconnections where the client nor the broker would detect the disconnection, five second disconnect time was used. TCP Proxy discarded all messages arriving during the five second period, after which it continued delivering messages between the MQTT broker and the subscriber client object. Even when it didn't notice the interruption in connection, the broker managed to deliver all messages and in the right order. All the messages accumulated to the broker during the disconnection were quickly delivered once the broker resumed delivering messages to the client object. Messages accumulated at the broker were usually delivered within 0.5 seconds after reconnection. Runs with slower results did manage to deliver most of the messages during the first 0.5 seconds, but spent 0.5 – 1.0 seconds overall delivering all the messages. After all the messages stored during disconnection were delivered, communication continued normally and messages were delivered similarly to what it had been before the disconnection.

QoS or other client side setting changes made no noticeable difference to the speed at which messages started flowing after reconnection, apart from the min and max retry interval for automatic reconnection. Figure 18 below shows results from a typical QoS 1 test run. “Default” represents a run with 20 seconds maximum reconnection interval and 10 seconds minimum reconnection interval for MQTT client’s automatic reconnection. For “short” those values were reduced to 2 seconds and 1 second. “Long” had values of 40 seconds and 20 seconds. Connection status for each run was either 100 or 0, 100 representing the connection being on and 0 being off. This way a clear picture could be provided when reconnection happened and when messages started flowing through. Figure 19 shows similar data for test runs with QoS changed to 2.

There was no real difference in functionality between different QoS settings. Both continued sending messages around the same time and delivered the stored messages within 0.5 seconds after reconnection. Variation seen in the figure 18 and 19 were caused by run to run differences and similar fluctuation was noted on same QoS settings between test runs. Reliability setting in MQTT client’s settings caused no changes in behavior either.

Figure 18 and 19 showed that smaller values in retry interval made a large difference. Default values, 20 seconds for max retry interval and 10 seconds for min retry interval showed results that varied between 4.0 seconds and 8.0 seconds. Most runs were closer to the 8.0 second mark, the kind of results seen in the figure 18 and 19. The ones that took less were noticeably closer to the 4.0, leaving an empty zone in between 5.0 and 7.0 seconds. When max retry interval was reduced to 2 and min was reduced to 1, it usually took only 2.0 seconds. Variance in results was a lot smaller and each run within 3.0 seconds from the reconnection. When the values were increased to 40 and 20 respectably, it took a long time for the communication between the broker and the client object to resume. These usually took around 17.0 seconds. There wasn’t much variety between runs and they were all within 2.0 seconds from one another.

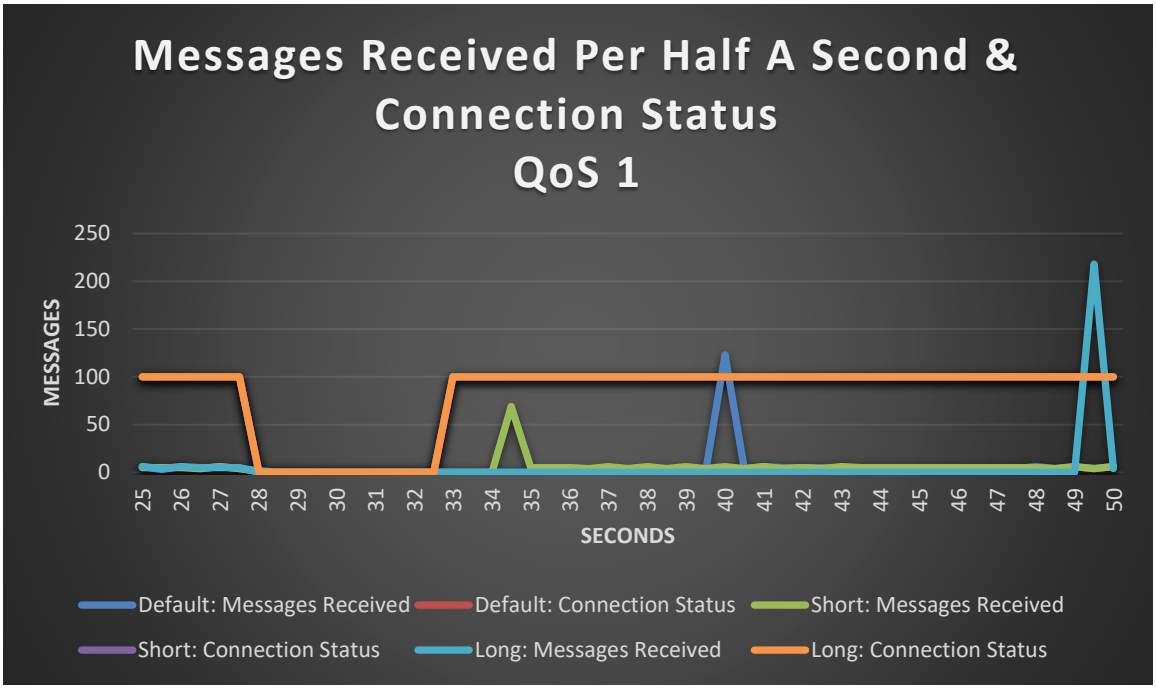


Figure 18. When disconnection went unnoticed, it took some time before the messages start arriving after reconnection.

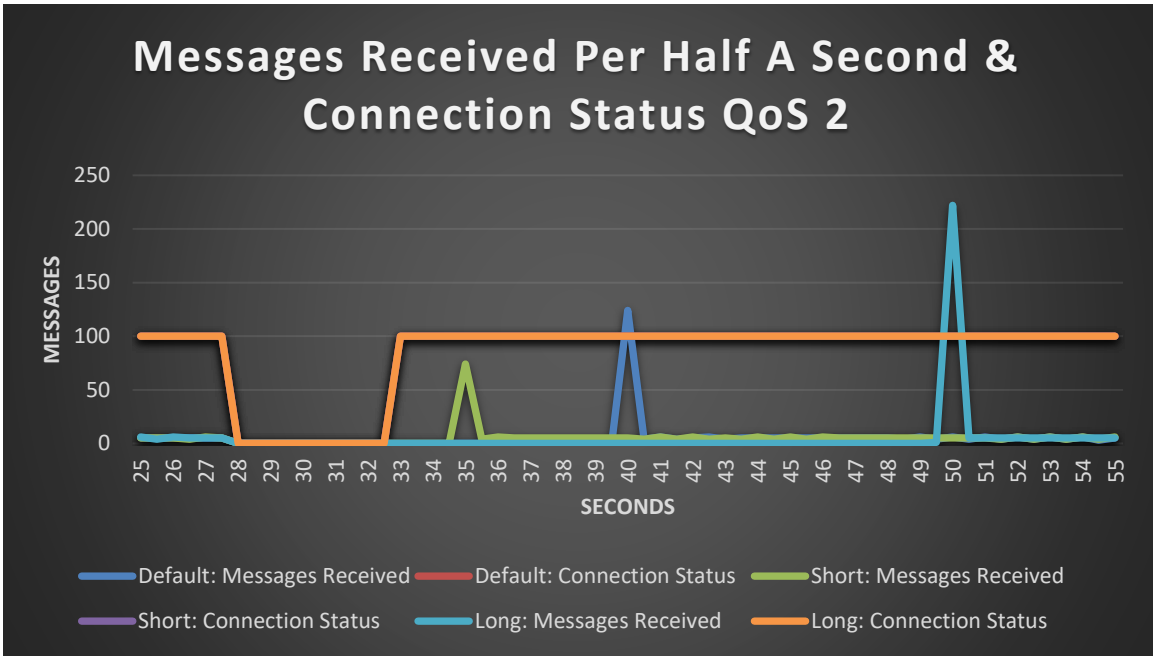


Figure 19. There was no noticeable difference between QoS 1 and QoS 2 settings.

4.2 Test case 2: realizing connection was interrupted

Testing long disconnection period where the subscriber client object and the broker would recognize the connection was interrupted was done similarly to case 1. The difference being that the disconnect time was increased to 30 seconds. Since the keep alive timer was not set higher than 20 seconds at clients' side, 30 seconds was enough for the client and the broker to realize the connection was down. Subscribing client object and the broker successfully handled long disconnections repeatedly. It took around 5 seconds for the client object and the broker to exchange messages again once the TCP proxy allowed messages to flow through. All accumulated messages were delivered within two seconds from the first delivered message. Most of the time one second was enough for all the messages to arrive. During testing this usually meant around 300 to 350 messages, but in cases where it took a long time to reconnect the amount was around 600 to 650 messages. After those couple of seconds where the broker delivered the missed messages, the message delivery normalized and the behavior was similar to how it was before disconnection.

Results from testing showed almost no setting from client's side made a difference in performance. Similarly to previous testing in Case 1, the only option that made a difference was the retry interval for automatic reconnection. As seen in figures 20 and 21 below, smaller the retry interval the smaller the delay between reconnection and continuation of delivering messages. Labels are the same as in Case 1, "short" meaning max retry interval of 2 seconds and min retry interval of 1 second. "Default" was 20 seconds and 10 seconds respectively. "Long" was 40 and 20 seconds for the intervals. Short managed to start within 0.5 seconds from the reconnection, which was different from Case 1 where the disconnection was not noticed. Default was a bit slower, usually starting to deliver messages within 3.0 seconds after disconnection. Long usually took way longer, taking between 30 and 35 seconds to start delivering messages.

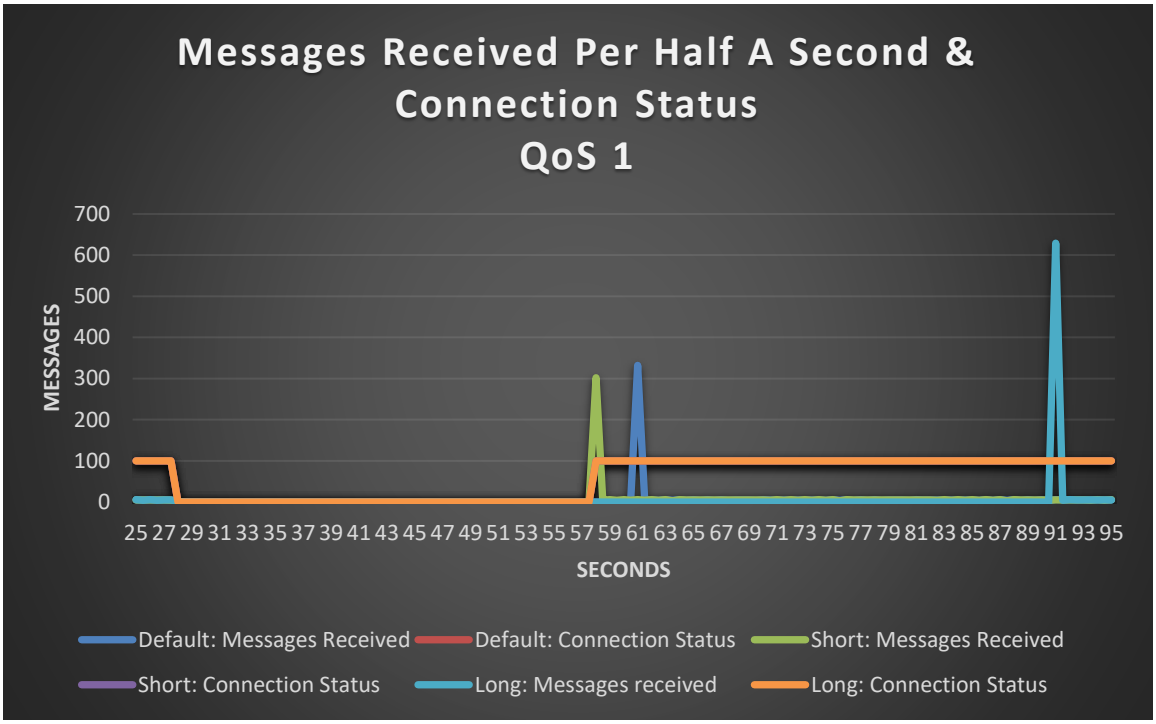


Figure 20. With short reconnect retry interval the broker managed to deliver messages within 0.5 seconds after reconnection was established between TCP Proxy and the broker.

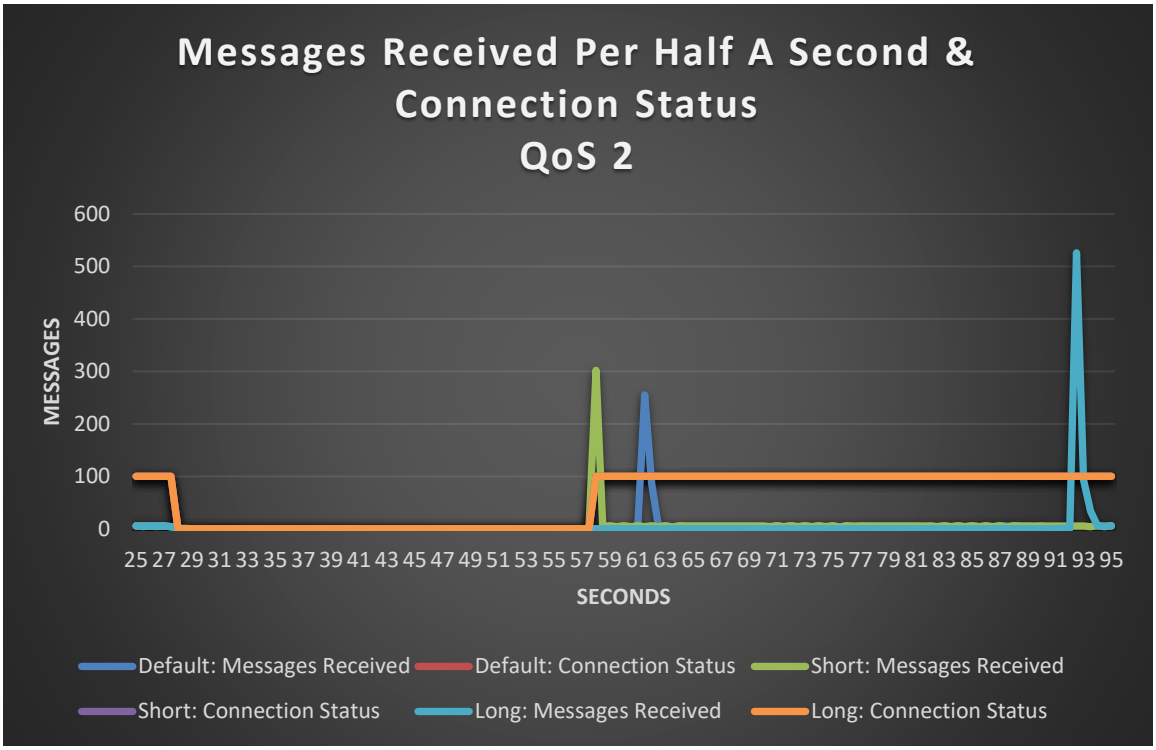


Figure 21. There was fluctuation in reconnection speeds between test runs. Some default test runs managed to reconnect as quickly as those with short reconnection retry interval, while being usually a little slower to connect.

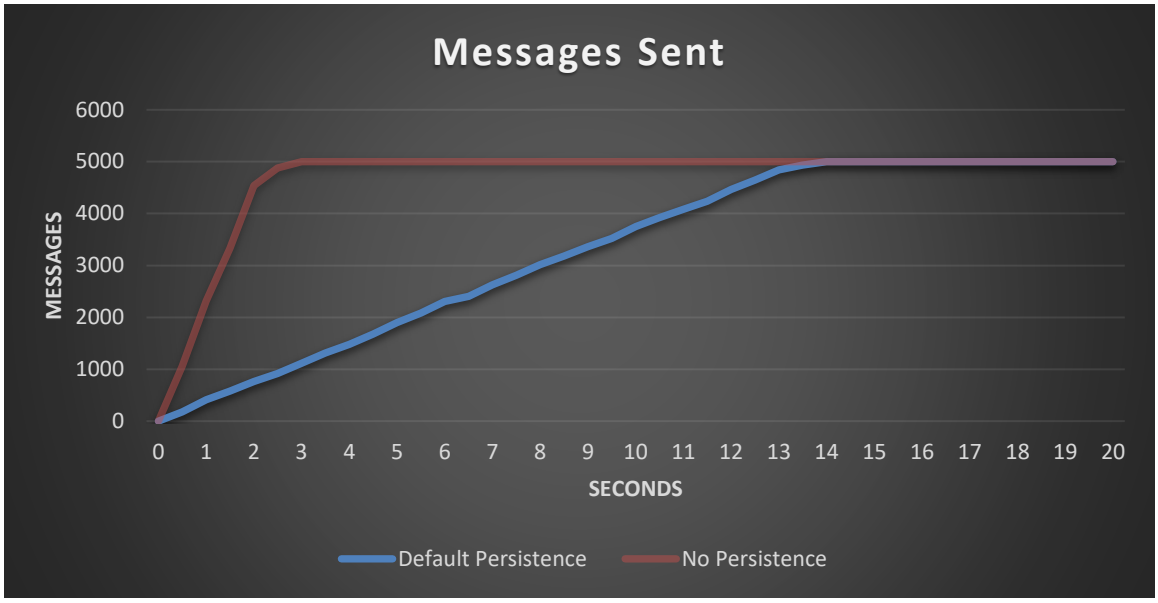
4.3 Test case 3: throughput

For throughput tests disconnection from the TCP Proxy was removed and the 100ms time between publishing messages within the publisher client object was dropped as well. Number of messages was also increased from 1000 to 5000. This was done to get clear results between different settings. MQTT broker's setting "max queued messages" was also increased to 5000. If the queue's cap was hit, it would have discarded old messages from memory to store more recent ones. Table 2 represents options that were tested for the broker and the clients. During a test run, both client objects had the same settings.

Client settings:	
Quality of Service	1 and 2
Clean Session	false
Reliable	20
Retain	1
Automatic Reconnect	1
Max Retry Interval	20
Min Retry Interval	10
Retry Interval	1
Persistence	false
Reliable	false
Broker settings:	
Max Inflight Messages	1 and 20
Persistence	true and false
Persistence: Autosave Interval	0,1 and 200
Persistence: Autosave On Changes	true and false
Max Queued Messages	1000, 3000 and 5000
Persistence File	set and not set
Persistence Location	set and not set

Table 2. Settings used to test throughput.

From the test results it was found that the default persistency is really slow compared to not using persistency on a client. Figures 22, 23 and 24 below compare the speed at which the clients were able to send and receive messages with the same settings, except from clients' side persistency turned on and off.



Figurer 22. Shows difference in sending speed between default persistence and no persistence. Results were at QoS1.

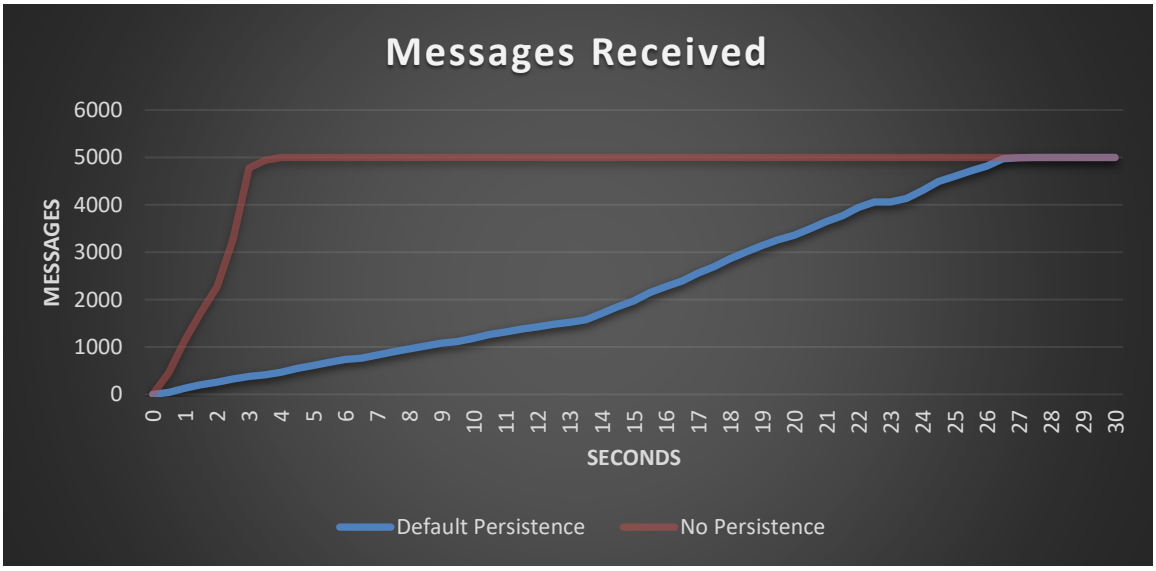


Figure 23. The difference in message receiving speed between default- and no persistence options. Results were at QoS1.

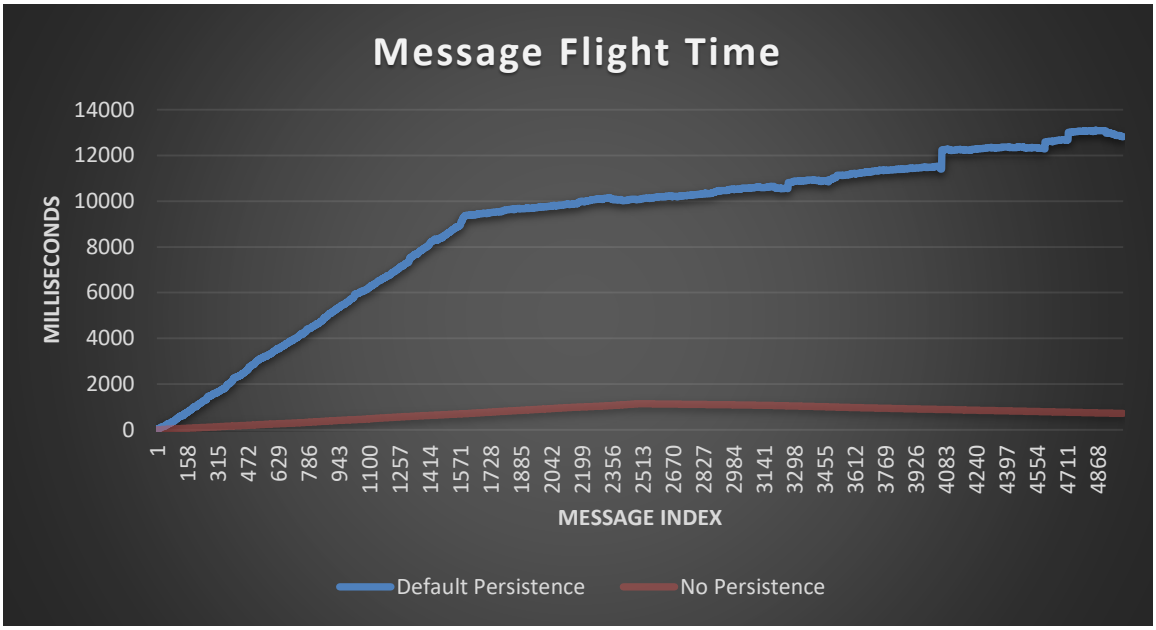


Figure 24. Flight times of each message with default persistency and without persistency. Results were at QoS1.

From the figures 22, 23 and 24, one could see the enormous difference in performance. Without any client side persistence it took between 2.5 and 3.0 seconds to send all messages with QoS 1, whereas with the default persistence it was between 13.5 and 14.0 seconds. On figure 23 there's an increase in receiving speed starting at 13.5 seconds for default persistence. But even with the increase, message flight time kept on rising. There was a really small decrease in the end of the flight time curve for default persistence. Flight time kept getting higher and higher despite the increase in receive speed. Sharp bumps in the flight time during the end of run with default persistence got even more prominent when the QoS was increased to 2.

QoS 2 throughput tests showed that broker's persistence also know as retained persistence option didn't seem to affect throughput in any way. Changes in message send speed were almost non-existent and are more likely caused by run by run differences than about the option affecting results. On message received speed there was a bit more differences starting from around 2000 messages received till the end. Changes there don't seem trustworthy. The difference between the runs with "max inflight" set to 1 seemed much larger than the difference found when max inflight option was set to 20. This suggests the difference here isn't reliable. It also only occurs on one spot and seem to flat out in the end between the options. Message flight time diagram even showed better results while persistence was set on. When max flight time was set to 1, the Message flight time had

less linear form. Instead some messages spent considerably more time inflight. Differences between Max inflight 1 option and max inflight 1 with persistence on option are hard to measure and are considerably more susceptible to run to run differences. Neither of the two was performing better than the other throughout the run suggesting persistence changes on broker had no effect on the results.

Figures 25, 26 and 27 below show run results with quality of service set to 2. All settings were identical apart from persistence and max inflight setting set within the broker.

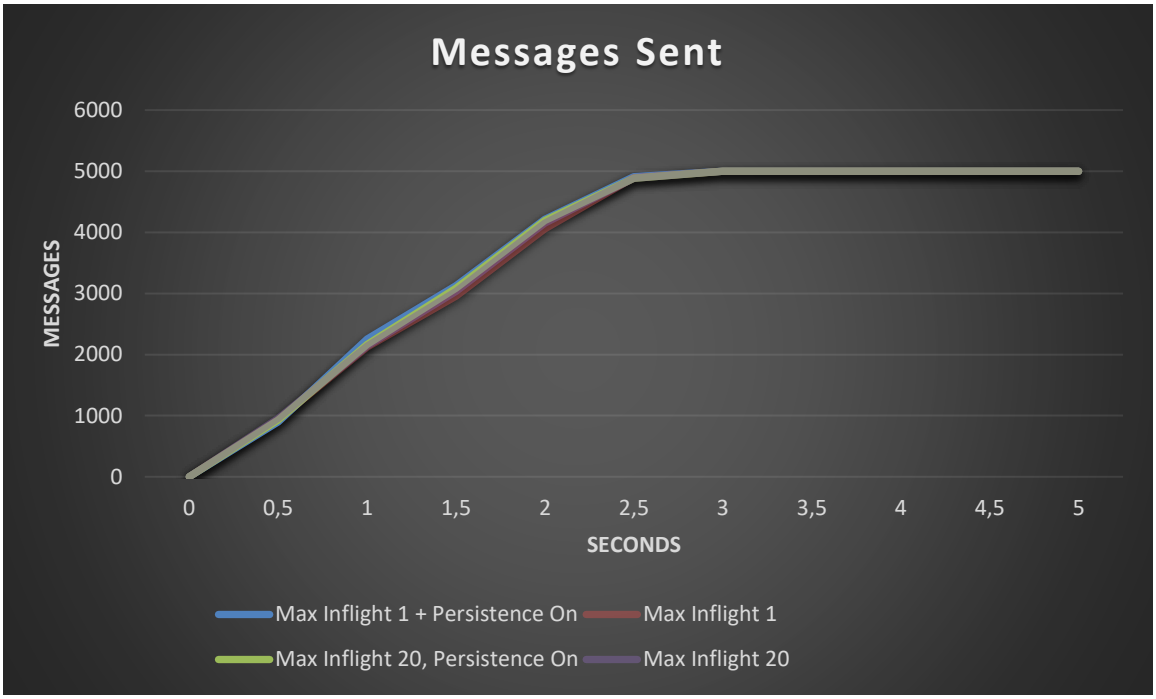


Figure 25. There was no clear differences between different tested options.

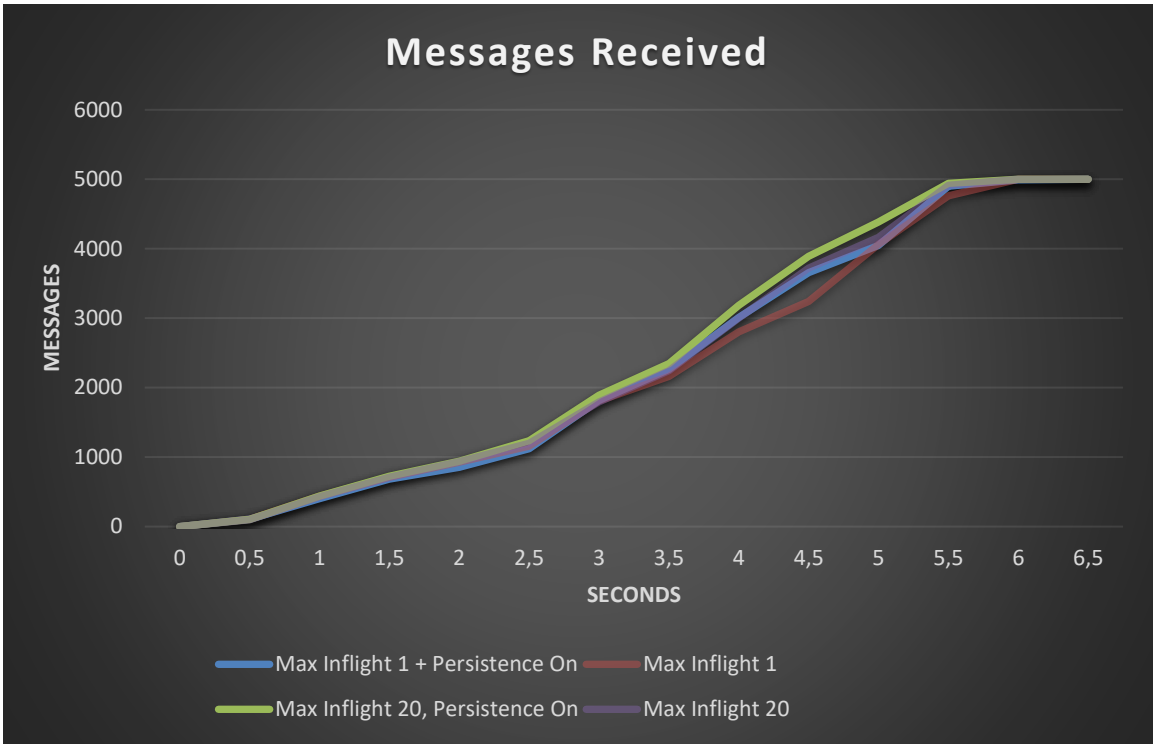


Figure 26. Small variation on results is due to differences between runs and not because one performed better than the other.

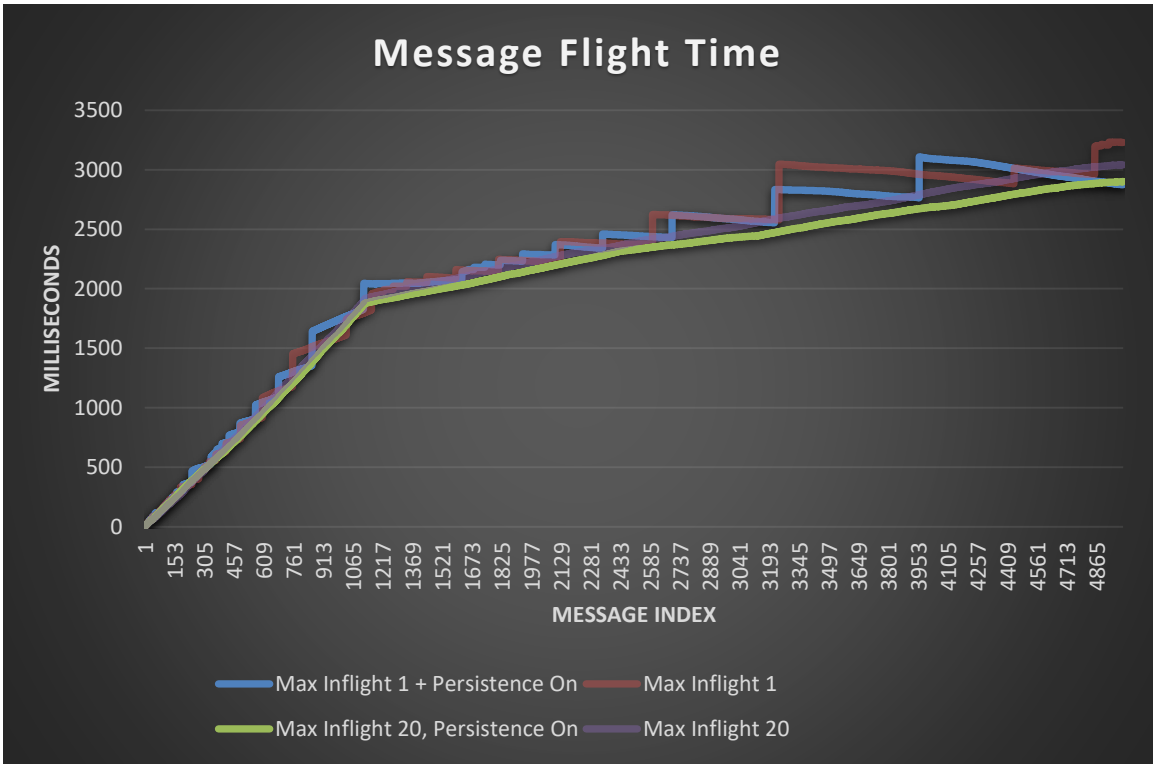


Figure 27. Test runs with option max inflight set to 1 gave uneven flight times between messages. Persistence setting seemed not to affect delivery speed.

While comparing QoS 2 runs, broker's persistence setting seemed to have no effect. Also gains from max inflight setting increased to 20 from 1 seem to be minimal. Overall they performed about the same on QoS 2. Broker's max inflight setting offered no performance gain during QoS 2 tests or the gain was minimal. Differences between runs were large enough that occasionally QoS 2 run with max inflight set to 20 did perform worse than a run with max inflight set to 1. Even though the receive speed keeps going up, there seem to be slowdown between 4.0 and 4.5 seconds. If there had been more messages, it's possible the speed could have risen, but considering it did slow down, it's possible the top speed for receiving for this setup was close to the speed seen here.

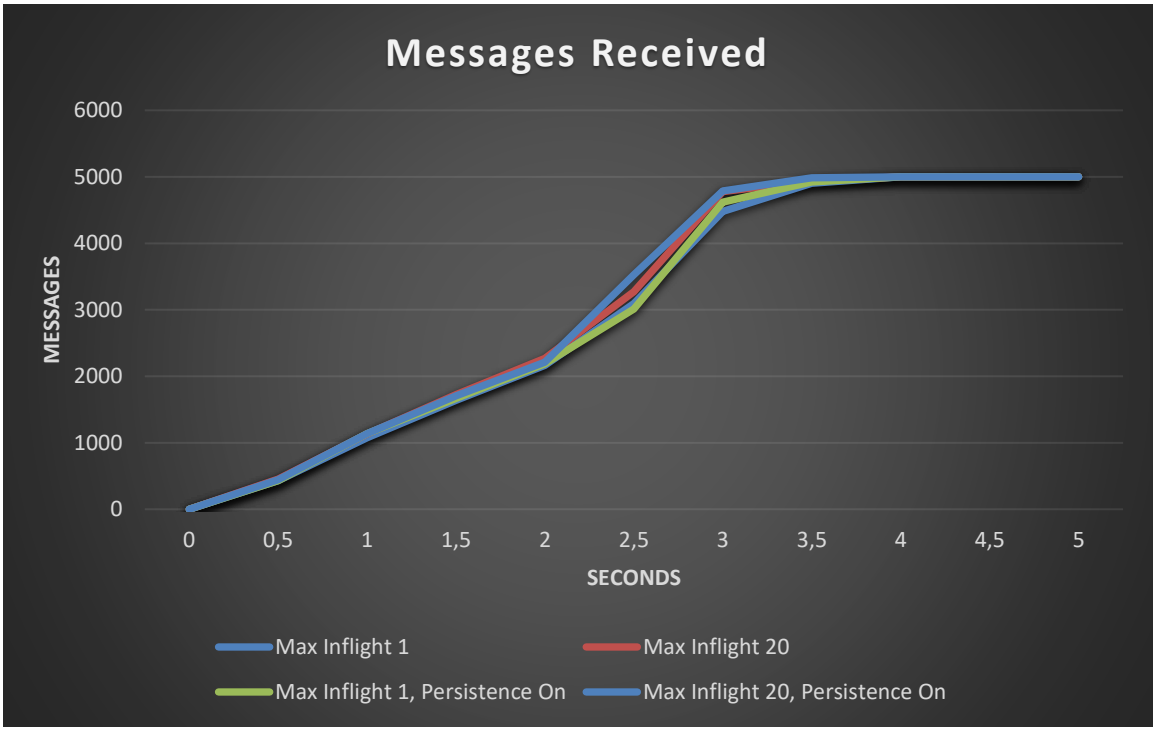


Figure 28. Difference between setting max inflight 1 from 20 offered no real performance penalty.

Figure 29. Message send speed was also around the same for each setting tested.

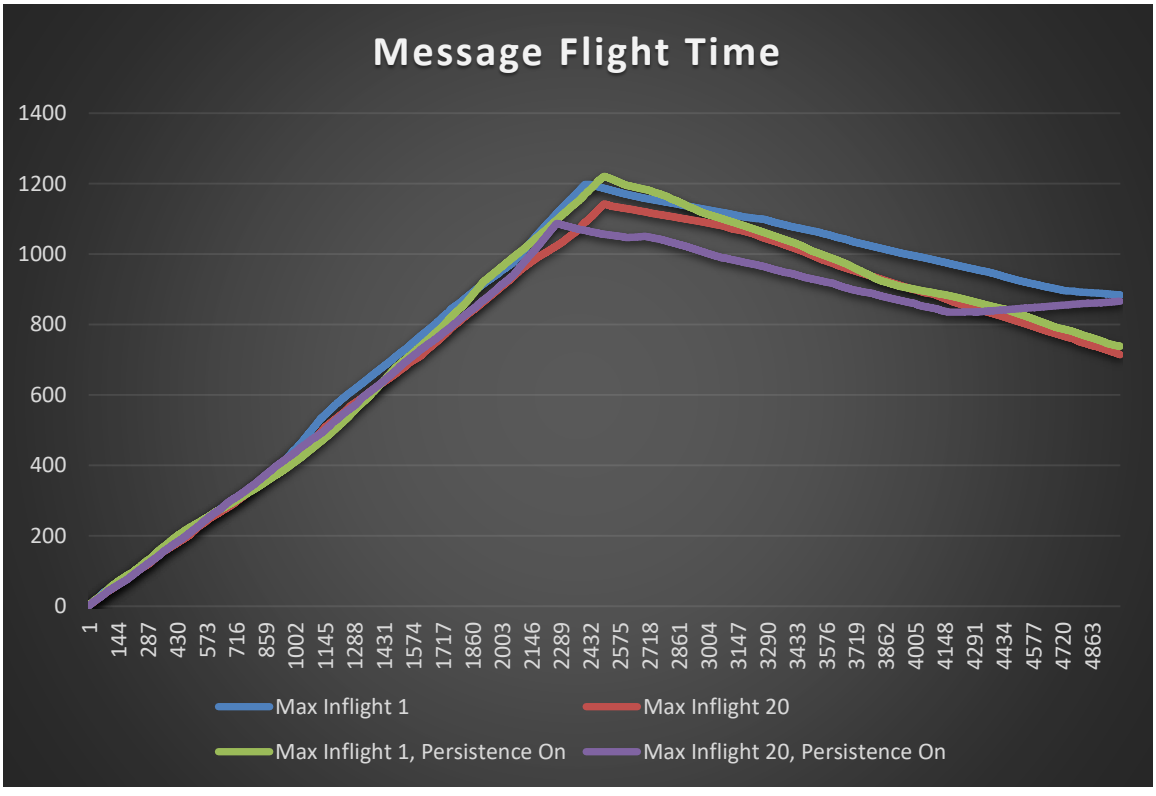


Figure 30. Peak in flight times is similar to each setting. After flight times start to decrease, flight times start to have more differences.

Figures 28, 29 and 30 above represent results received from QoS 1 test runs. Overall performance was higher than what it was with QoS 2, but differences between different QoS 1 settings proved to be minimal. Message receive speed increased at 0.5 seconds for the first time, offering a slightly faster delivery speed. Around 2 and 2.5 seconds there was another increase, this time more noticeably. At this point message flight time started to decrease. QoS 1 kept the delivery speed about the same until almost all the messages had been delivered, at which point it slowed down again. If there had been more messages it's possible that the delivery rate would have gone up since there was no slow down until the very end.

Figure 31. With QoS 1 client was able to send messages a bit quicker. Both runs in both QoS classes performed very similarly to each other.

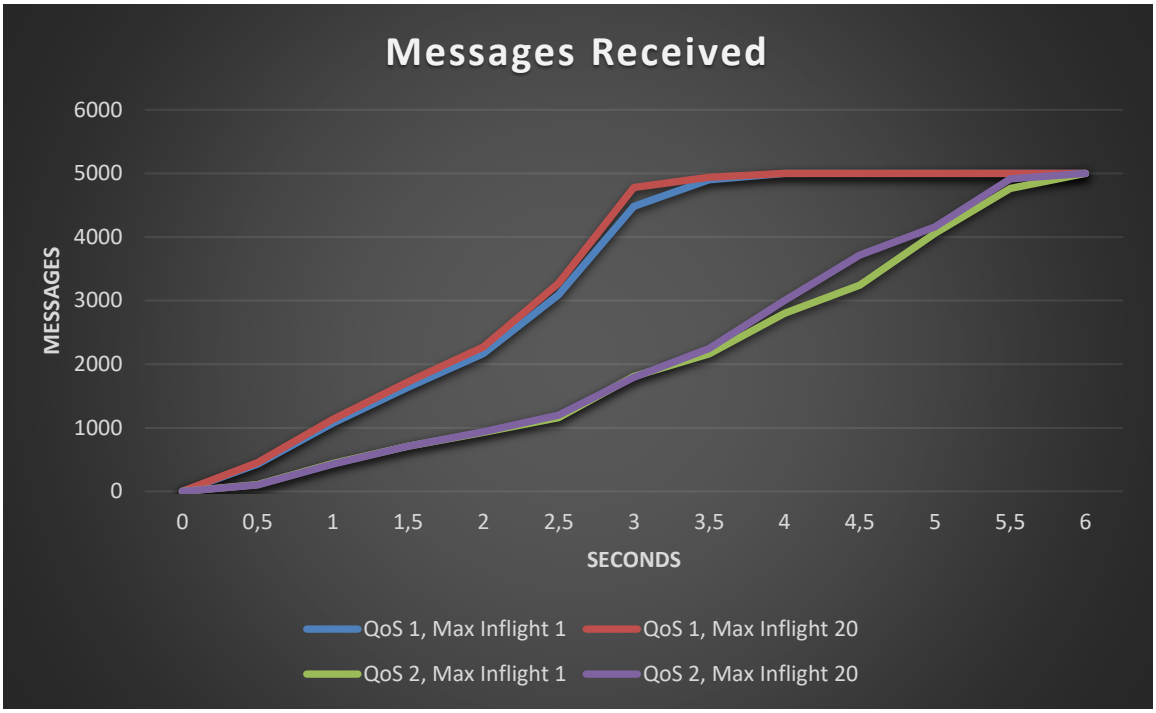


Figure 32. QoS 1 offered faster received speeds than QoS 2.

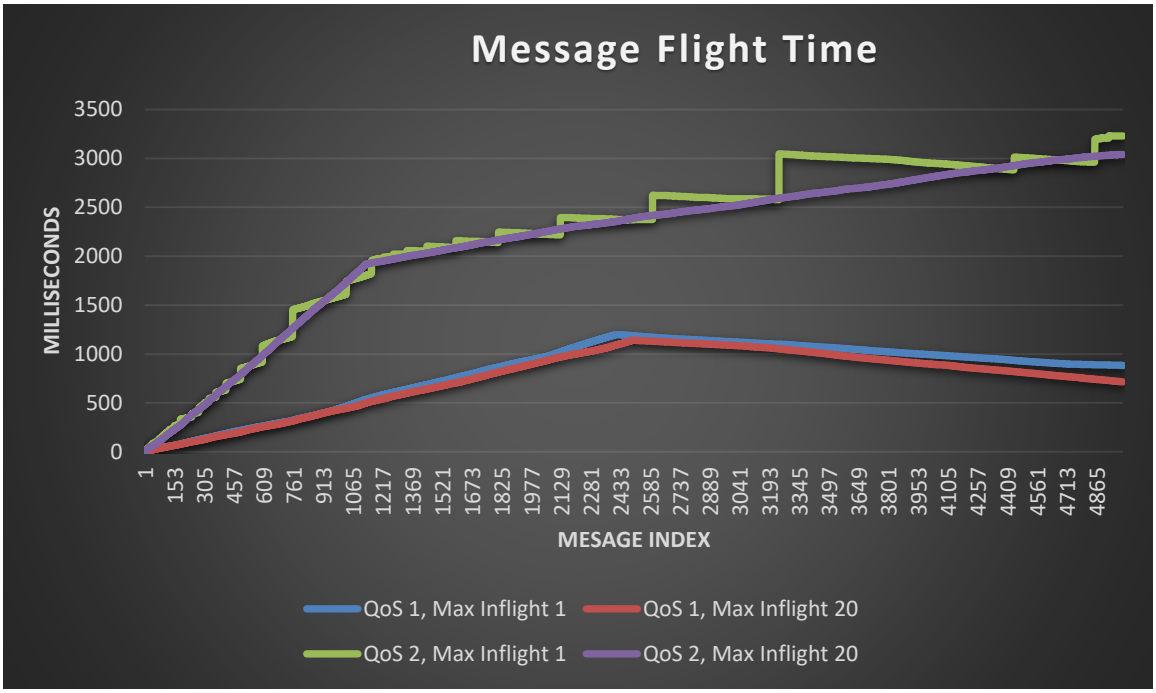


Figure 33. Transfer rates were higher with QoS 1, offering a lot lower flight times for each message. With QoS 2 flight time kept getting higher and higher indicating that the delivery couldn't keep up with the send speed. QoS 1 however managed to peak around half way and was able to reduce delivery time from there.

Figures 31, 32 and 33 compare QoS 1 and QoS 2 runs to each other. Even though send speed was similar between the two, there was a large gap between them when comparing the speed at which they received messages. QoS 1 managed to outperform QoS 2 with almost 50% faster overall receive speed. On figure 33 above the flight times for each compared run were shown. There was no indication that QoS 2 could reduce the flight time if more messages had been sent, whereas with QoS 1 there might still be a little more flight time that could have been cut. Even if QoS 1 managed to keep the current speed, QoS 2 would most likely fall behind more and more with its raising flight times.

Tests to see the performance impact of the broker's persistency options were also performed. At first it was examined how persistency autosave interval would impact performance. On the client side, every other setting was set as the same, apart from QoS. Tests were performed on both, QoS 1 and QoS 2. First tests to see whether or not it affected performance were run with persistency autosave interval set to 200. This way it shouldn't save anything on disk during the test. This was used as a baseline for results gathered from runs with persistency autosave interval was set to 1. Persistency's autosave functionality was disabled and persistence file and locations were left empty. Test results showed that differences for QoS 1 and QoS 2 were miniscule and within a margin of error. There were no clear patterns that would differentiate runs with autosave interval set to 1 and the ones where it was 200. They both also received all messages around the same time, in most runs within the same 0.5 second time step.

Then broker's autosave on changes functionality for persistency was tested. For QoS 1 the difference was huge. Messages were sent at the same rate, both succeeding to send all messages within a second from one another, but receive times were different. Without autosave, all message had arrived within 4 seconds from the start. But with the autosave on change functionality enabled, this time was increased to 7.5 seconds. The sharp sloping towards the middle of the "messages received" line that had been common amongst results gathered was not there when autosave was enabled. Instead there was a very gradual increase with a part where messages were not received at all. Similar behavior was seen on another test run, but third test didn't have it. It seemed somewhat irregular behavior. Interestingly once persistence file was set to mosquito.db and persistence location was set to C:/Users/extluomaal/Desktop/thesis/Release/, it changed around. With those set, the difference disappeared. Suddenly messages arrived as quickly as they would without the "autosave on change" options set on. According to Mosquitto's documentation [16], Mosquitto should provide a default location on if persistence location is left empty, which is supposed to be "current directory". A quick search found mosquito.db

from within windows folder, which was on the same SSD as the folder where the new location was. Mosquitto was installed as a windows service on test machine. Since it was operating on the same SSD on both cases, it seems unlikely it would be drive related. Tests were run as an administrator and without administrator rights to see if that would change things but it didn't. So In the end it was left unclear why these results were received. Figure 34 below shows the results in message receive speed from the change in the settings mentioned above.

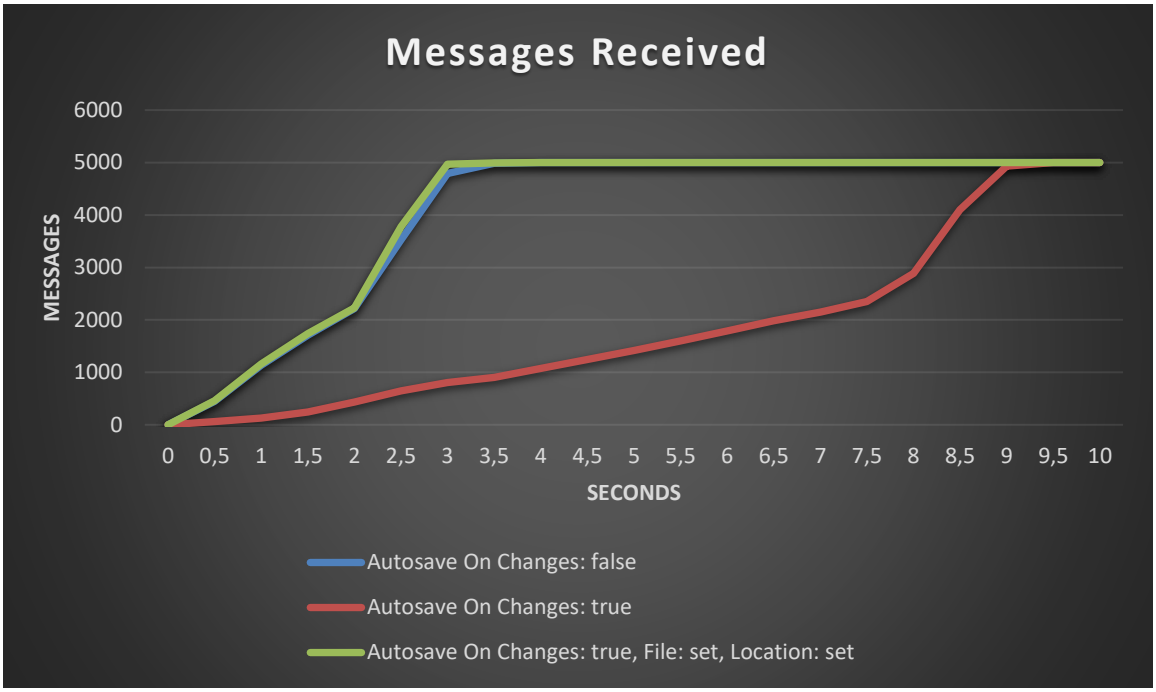


Figure 34. When the database file name and database location weren't set, autosave on changes setting gave slower results.

Without the autosave functionality the flight time starts to decrease once 2000 to 2500 messages had been delivered. Flight time peaked at around 1200ms. But once the autosave function was enabled, the results showed that the message flight time kept growing. Since flight times kept increasing steadily, it suggests the peak was not hit and with a higher message count the flight times could keep increasing even further. This was behavior disappeared completely when persistence file and persistence location options were set from within Mosquitto's "mosquitto.conf" configuration file. Figure 35 below shows the differences in flight times.

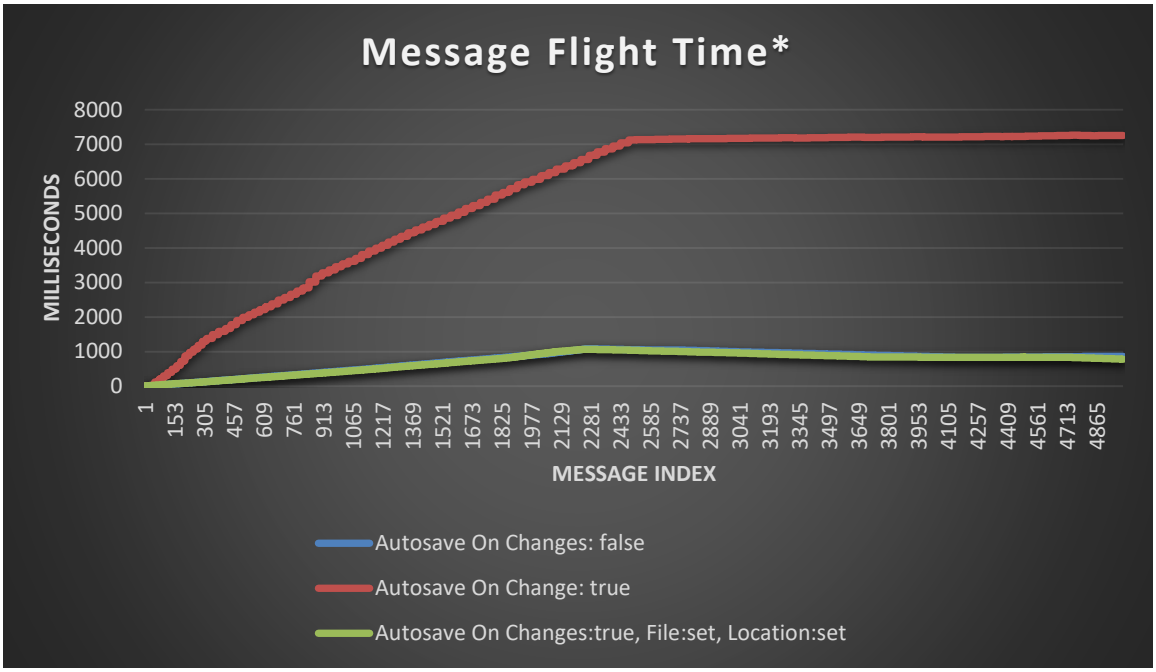


Figure 35. When database file name and location weren't set, Atuosave on Changes behaved completery differently. When they were set, it behaves just like it did without the whole autosave functionality. Without them message flight times kept climbing up while autosave on changes was enabled.

5 Summary

Tests showed that a MQTT client settings for the most part didn't have any effect at how the client recovered from a disconnection. Results from tests showed that message delivery continued quicker when the disconnection time was longer and clients were able to determine that the connection had been interrupted.

Apart from min and max retry interval, different options didn't seem to have an effect at how quickly MQTT broker continued sending messages to the subscriber client object. The lower those two were the better the results. But dropping down retry interval values from 20 and 10 seconds to 2 and 1 second achieved only small gains. Time to resume sending messages was reduced from 3.0 – 5.0 seconds to 0.0 – 3.0 seconds. When min and max value were increased to 40 and 20 seconds, the reconnection time was increased greatly. It took anything from 25 to 35 seconds to reconnect in both disconnection test cases.

Throughput tests were mostly affected by QoS and persistence settings. While delivering 5000 messages with QoS 1 took around 2.5 to 3.0 seconds, QoS 2 took from 4.5 to 5.5 seconds. When default persistence was used, all but QoS 2 with max inflight set to 1 were really similar in terms of performance. These tests runs took from 24 to 27.5 seconds to deliver all messages. QoS 2 with max flight set to 1 spent roughly 32 seconds delivering the same amount of messages.

When autosave on changes and persistence broker settings were on and broker's persistence's autosave interval set to 200, throughput tests did show unexpected behavior. Messages with QoS 2 and max inflight setting set to 1, it took around 23 seconds to deliver messages. A few messages arrived in the beginning, then the flow stopped. Only after 6 seconds of no messages coming through since the initial messages, more messages finally started arriving. When max inflight was increased to 20, QoS 2 delivered messages in around 18 seconds. The slowdown on the start also took less time, around 4.5 seconds. QoS 1 didn't have similar issues with not sending messages for a little while at the start of a test run. QoS 1 with max inflight set to 1, managed to deliver all messages in 9.0 seconds. When max inflight was increased to 20, delivery time was reduced to anything between 6 to 8 seconds. Multiple runs did land closer to the 8 seconds than 6 seconds.

When “autosave on changes” setting was on with persistence’s “autosave interval” setting set to 1, throughput tests failed. They were either unable to deliver all messages, or they delivered some messages multiple times, even with QoS 2.

6 Things to improve

Test setting could have been improved in many ways. During testing the point at which the time was recorded was moved as it wasn't initially right after the connection. This caused some inaccuracies which forced me to retest reconnection cases. This could have been avoided with better planning. I also didn't use the same line of settings for both clients from the very start which turned out to be a hassle. Every now and then a setting here or there was left unchanged when it needed changing, causing wasted time as the test result became unusable. Parts of the test setup were added in the beginning when I thought they could become handy, but turned out to be a waste of time in the end. Like setting default parameters. I also didn't factor in the time I'd spend creating the logging system while planning time management for the project.

Even though it was working well and provided a document that was easy to export to excel, it was missing settings the information for mosquitto's configuration file. This made it unnecessarily tedious to keep up which settings were being tested when multiple tests were run in short time. A better way would have been to read the configuration file and wrote all the lines that weren't commented to a separate part of the log file. Another thing that could have been handy would have been a system to fill up a table where I had all the test runs and their settings stored. Filling it by hand took some time and allowed error to sneak in. This also forced me to redo some tests when I inserted a wrong file name to the file. I was happy with system otherwise. Also some poor naming ended up causing me to create another column for information already found from within the document.

Within the client a there were tiny things that could have been improved. More commands could have been sent between the proxy and client in the beginning instead of relying on a timer based system. This would have improved productivity when systems wouldn't have had to wait when both were ready. Same is true for closing down the systems. When subscriber was ready, it should have sent a command to proxy, telling it to shut down. The communication part could have been revisited as well as the performance wasn't best possible. With QoS 0 I was able to get messages through in 0.37ms at best. It was enough to be able to compare differences between runs but throughput tests would have given better results with some updating. There were unnecessary checks that could have been avoiding with better design. Also adding C++ version of the MQTT client would have been preferable

7 Conclusion

Since the reconnection always happened around the same time for each test in each test case, it's possible they hit some sweet spot for the client's reconnection check. The default option should only reconnect after 10 seconds, yet it repeatedly connected well below that. Similarly it's possible when the reconnection interval values were increased to 40 and 20 seconds, it missed some critical time point for the check. By multiplying both reconnection interval values by two, one would expect linear increase in reconnection time, but the default gave many times faster reconnection.

For throughput the QoS had a noticeable difference in performance. This was to be expected due to increased network traffic. Default persistence didn't show performance penalty when connection was simulating a real world device with its 100ms delay between messages, but in the throughput tests it was really slow in comparison. Being 6 to 7 times slower is very noticeable and could very well end up being a problem in a big scale system.

Max inflight had some effect to throughput, causing stuttering in some tests for flight time. Overall performance stayed about the same when max inflight was set to 1 compared to the default 20.

Results received from testing broker's persistence's autosave setting suggest there was a bug in Mosquitto MQTT broker's software. When persistence file and location were set, performance was much better suggesting there's something not right there. They shouldn't affect performance in any way since they both should have a default value in case they weren't provided.

Since tests were ran over multiple days, there could be differences due to system updates or something similar. A new version of Paho's C client was updated to version 1.2.1 during testing which could have improved performance, but its testing was not included in this thesis. Tests were ran multiple times for each test to get an indication whether or not the results were aligning with each other. If they weren't, a test setting was run more than twice to get ensure the results weren't caused by problem with hardware of software outside the test setting.

Sources

[1] [Internet]. Cited 16 April. Available from: https://www.tutorialspoint.com/data_communication_computer_network/transmission_control_protocol.htm
[2] [Internet]. Cited 21 May. Available from: https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[3] [Internet]. Cited 16 April. Available from: http://mqtt.org/faq
[4] [Internet] MQTT Version 3.1.1 Standard documentation pdf, page 1. Cited 17 April. Available from: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf
[5] [Internet]. Arlen Nipper. Cited 18 April. Available from: https://www.linkedin.com/in/arlen-nipper-42281057
[6] [Internet]. Cited 17 April. Available from: https://www.oasis-open.org/news/announcements/mqtt-version-3-1-1-becomes-an-oasis-standard
[7] [Internet]. Cited 17 April. Available from: https://www.iso.org/standard/69466.html
[8] [Internet]. Cited 18 April. Available from https://github.com/mqtt/mqtt.github.io/wiki/things
[9] [Internet]. Written by Karen Lewis. Cited 18. April. Available from: https://www.ibm.com/blogs/internet-of-things/andy-stanford-clark/
[10] [Internet]. Cited 19 April. Available from: https://github.com/mqtt/mqtt.github.io/wiki/servers
[11] [Internet]. Cited 19. April. Available from: https://www.ibm.com/developerworks/library/iot-mqtt-why-good-for-iot/index.html
[12] [Internet]. MQTT Version 3.1.1 Standard pdf. Cited 2 May. Available from: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf
[13] [Internet]. Cited 3 May. Available from: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels
[14] [Internet]. Cited 18 June. Available from: http://www.admin-magazine.com/Archive/2015/30/Using-the-MQTT-IoT-protocol-for-unusual-but-useful-purposes
[15] [Internet]. Cited 20 April. Available from: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels
[16] [Internet]. Cited 21 June. Available from https://mosquitto.org/man/mosquitto-conf-5.html

Appendices

Setting up a Mosquitto broker

Mosquitto broker was selected to be used while working on thesis, as it is an open source MQTT broker and it's supposed to be lightweight. It was also pretty straight forward to setup and offered enough documentation to make it easy to use.

1. Setting up

To setup a mosquitto broker, a couple of things were required. Mosquitto version 1.4.15a was used for the thesis. A ready binary installation can be obtained here: https://mosquitto.org/download/. Once it was downloaded and installation started, it requested to download pthreadVC2.dll and Win32 OpenSSL. Pthread version used in for the thesis was obtained from ftp://sources.redhat.com/pub/pthreads-win32/dll-latest/dll/x86/. For Win32 OpenSSL version 1.0.2o Light was used which can be obtained from http://slproweb.com/products/Win32OpenSSL.html. At first OpenSSL version 1.1.0h Light was tested, but it didn't provide the files needed for Mosquitto. Ultimately version 1.0.2o Light was used, which did contain all the necessary files. OpenSSL's installation path for the thesis was "*F:\thesis\OpenSSL-Win32*". After installing Win32 OpenSSL, Mosquitto was installed. For the thesis, installation path "F:\thesis\\Mosquitto" was used. Once the Installation was completed, a couple of files needed to be copied from the Win32 OpenSSL folder to the Mosquitto folder. Following DLLs were copied; ssleay32.dll and libeay32.dll to the mosquitto folder. Also the previously downloaded pthreadVC2.dll was also copied to the Mosquitto folder.

2. Testing broker

Command prompt was opened, inside which it was navigated within Mosquitto folder. For the thesis command F: was used, followed by cd F:\thesis\Mosquitto. F: allowed to change drive and cd to get inside the Mosquitto folder. Inside the Mosquitto folder, mosquitto.exe was ran. When it was running, two more command prompts were opened. Inside the command prompts the same Mosquitto folder was opened. To test if the mosquitto.exe was working properly a mosquitto_sub.exe –t messages command was executed in one of the previously opened command prompts. Mosquitto_sub.exe is a client provided by mosquitto. Command line argument –t indicates a topic was given to the subscriber, followed by "messages", which in this case was the name of the topic. Next in the last of the three command prompts a command mosquitto_pub.exe -t messages -m message was executed. This opened a publisher client. –t indicates we gave it a topic, named "messages" in this case. –m indicates a message was provided, in this case that was "message". Subscriber received a message,

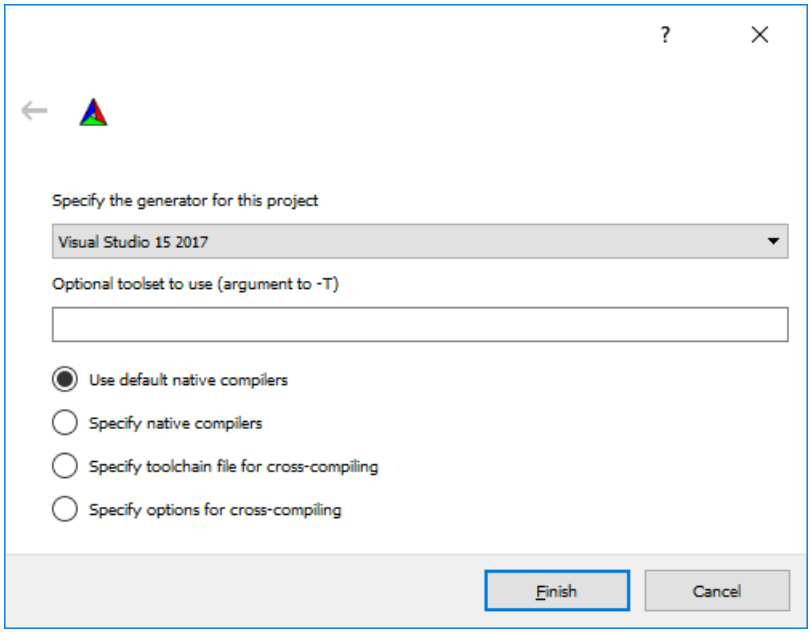
informing everything was working correctly. Once it was made sure the Mosquitto broker was working properly, Mosquitto broker were turned into a windows service by installing it again in the same path. After installation it was checked there was a service named Mosquitto Broker. Inside windows 10 services tab could be accessed by typing “Services” in the start menu search bar. It was ensured the broker’s status was “running”. If it hadn’t been it could have been turned on by right clicking Mosquitto Broker service and pressing “start” from the pop up options. By right clicking and going into the property tab one could change broker’s startup type. Automatic would turn it on when the windows starts whereas manual would require one to manually start the broker every time windows was turned on.

Setting up Paho C Client

There is a github page for setting up Paho MQTT C client which offers a great documentation how to set things up. Version 1.2.0 was used in the thesis.

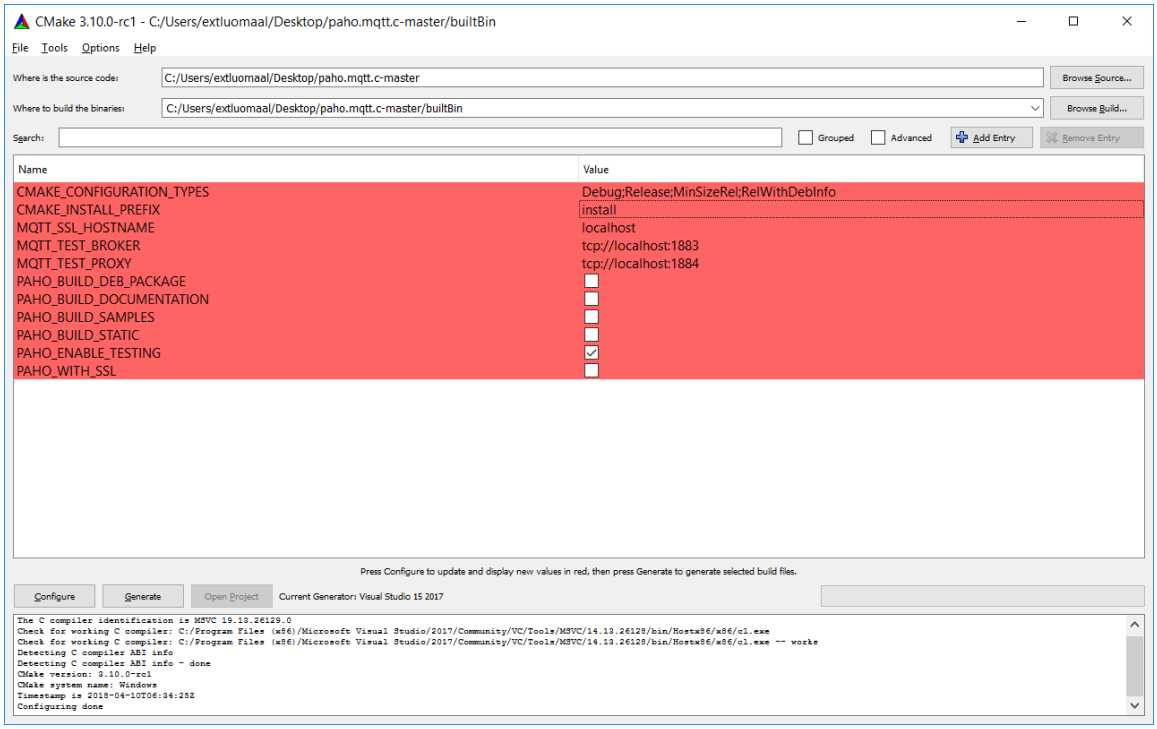
1. Libraries and headers

First step was to create a folder called “dependencies” at the same location where the .sln solution file was located that contained both, TCP proxy and MQTT client. Inside that directory a new folder called “mqtt_c” was then created. Then MQTT C client for Windows was downloaded. Client’s source code was obtained from https://github.com/eclipse/paho.mqtt.c. Zip file was downloaded and un-packed into a folder on Desktop. Afterwards CMake GUI was opened. Browse for Source code was pressed and a directory where binaries were built was selected. Configure was pressed. Directory for binaries did not exist and Cmake was allowed to create a new folder. A generator pop-up windows appeared afterwards and Visual Studio 15 2017 was selected. No optional toolsets were used and “Use default native compilers” was selected. Picture 1 below shows the selected options.



Picture 1. Native compilers were used for Visual Studio 15 2017

A couple of configurable options appeared on CMake GUI’s main window. CMAKE_INSTALL_PREFIX was changed to “install” so could be found from the same place as the rest of the files generated. Other options were not touched. Following picture 2 shows the options



Picture 2. Cmake options for Paho's MQTT C client.

Afterwards build files were generated. When CMake was done generating build files, Project was opened by pressing "Open Project" in CMake's main window. From within Visual Studio INSTALL project was then built. Within the directory created for binaries, buildBin in this case, install folder could be found. Inside that there was a folder called bin, inside which include and lib directories could be found. Lib and include folders were copied to the mqtt_c directory which was created earlier.

Solution file where the MQTT client and TCP proxy were created was opened next. From within the solution MQTT client and TCP proxy projects' properties were configured. Inside C/C++ tab, in "General" sub tab following path was added to Additional Include Directories field: $(SolutionDir)dependencies\mqtt_c\include. The same line was also added to "Additional #using Directories" field. This way it was not required to type the full path whenever something was needed from inside the folder. Inside the "Linker" tab's "General" sub tab following path: $(SolutionDir)dependencies\mqtt_c\lib was inserted to "Additional Library Directories" field. Inside the "Input" sub tab paho-mqtt3a.lib and paho-mqtt3c.lib libraries were added to the "Additional Dependencies" field. Finally inside "Build Events" "Post-Build Event" sub tab was opened. Following command was entered to "Command Line" field: *XCOPY $(SolutionDir)\Dependencies\mqtt_c\lib*.DLL "$(TargetDir)" /K /Y.* Following text was added to the Description field: "Copy DLLs to Target Directory." Given command copied all .dll files from dependencies folder using wildcard, to project's build folder when it was built. Inside "Configuration Properties" tab's sub tab "General", *.dll was added to the "Extensions to Delete on Clean" field. This

way all files will be removed from the build directory that are being inserted there when the project is built. Otherwise all .dll files that were copied inside the build directory would be left behind by the cleaning up process.

2. Testing libraries and headers

To test the setup, paho-mqtt c client example was copied for testing. Code was obtained from: https://www.eclipse.org/paho/files/mqttdoc/MQTTAsync/html/subscribe.html. To be able to successfully build the example windows.h needed to be included or WIN32 and WIN64 macros wouldn't be recognized. It was also required for sleep function. static casting for char was added where payloadptr was set as message's payload inside msgarrvd function.

Command prompt was opened and a command F: was executed, followed by a command cd thesis. Once thesis folder had been reached, paho's mqtt c client was cloned using git commands. Command "git clone https://github.com/eclipse/paho.mqtt.c.git mqtt_c" was executed. Last part of the command "mqtt_c" told which folder the git branch should be copied to. This way files were kept neatly packed. After this, command "cd C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\MSBuild\15.0\Bin" was executed. If some other than Community version of Visual Studio would have been used, the path would have required adjusting. From the MSBuild's binary directory command: MSBuild.exe "D:\thesis\mqtt_c\Windows Build\Paho C MQTT APIs.sln" /p:Configuration=Release was then executed. Here an error was provided about not having The Windows SDK version 10.0.14393.0. This was fixed by running Visual Studio Installer, where under "Universal Windows Platform development" tab the Windows 10 SDK (10.0.14939.0) version were added by ticking the corresponding box, and then pressing "Modify" from the right bottom corner. Paho's C Client was now ready to be used.

Setting up SFML in Visual Studio 2017

Simple and Fast Multimedia Library, known as SFML for short. It's a multimedia library which offers easy to use modules for 2D graphics, Networking, Sound and Window handling. It's mostly used by small game and/or indie developers. Its user base is relatively small but growing. The first version of the Library was release on 9 August 2007 and it has been updated regularly ever since [1]. Current version during the writing of this document is 2.5.0 [2]. SFML was chosen for its ease of setting up and using while providing good documentation were something to go wrong. The network module was used for this project.

1. Setting up SFML libraries

Since the project was built on Windows as a 32-bit application, binaries were pre built and were obtained from SFML's website: https://www.sfml-dev.org/download.php. Because there were two projects within the solution, proxy and client, these steps were done to both of them. Adding SFML packet as part of a project is straight forward and fully fleshed out tutorial for visual studio can be found from https://www.sfml-dev.org/tutorials/2.5/start-vc.php. Firstly include directories were added, followed by setting linker options. They were set on both, debug and release builds. *$(SolutionDir)* macro was used which allows the folder where the .sln file lays to be used as the starting point for the folder path. It's noteworthy that $(SolutionDir) requires no "\" at the back of it.

Next step was to add Additional Dependencies within Linker/Input page. Only sfml-system and sfml-network were added since graphics, window nor sounds were not needed. Since sfml-system and sfml-network have their own dependencies, those were added as well. Final list of all the files added to dependencies can be found from a picture below. This allowed to use the network features of SFML library.

2. Testing SFML libraries

To make sure the SFML library were working correctly, Network.hpp from SFML was included to the client's main.cpp. A single sf::Thread object was added inside main.cpp and the code was compiled. As it succeeded everything was set correctly for SFML network module. The same procedure was done for the proxy project.

3. Sources

[1] [Internet]. Cited 10 May. Available from: https://en.wikipedia.org/wiki/Simple_and_Fast_Multimedia_Library

[2] [Internet]. Cited 10 May. Available from: https://www.sfml-dev.org/download.php